P148505.PDF [Page: 4 of 108]

905450
904576

94-627

AN INVESTIGATION OF SPECIAL PURPOSE HARDWARE FOR FINE CHANNELIZATION IN THE AMEP PREPROCESSOR

Contract No.: 66ER-1-0312/5C-ER

Document No.: 1005058
Issue -

Date: 23 March 1994

Prepared for:

Defence Research Establishment Ottawa
Shirleys Bay, Ontario

ROBERT INKOL
SCIENTIFIC AUTHOR

Prepared by:

LOCKHEED CANADA INC.
1 Iber Road
Stittsville, Ontario
K2S 1E6

P148505.PDF [Page: 5 of 108]

APPROVAL RECORD

FOR

AN INVESTIGATION OF SPECIAL PURPOSE HARDWARE

FOR FINE CHANNELIZATION IN THE

AMEP PREPROCESSOR

Prepared by: _____ March 25, 1994
Jim Mickeal
Principle Engineer
Date

Prepared by: _____ Mar 28, 1994
Gordon MacKean
Senior Hardware Engineer
Date

Released by: _____ 28 Mar 94.
G. Gagne
Configuration Management Office
Date

P148505.PDF [Page: 6 of 108]

DOCUMENT CONTROL LOG

ISSUE NO.	DATE	REASON FOR CHANGE
-	23 March 1994	Initial release

ISSUE RECORD

SHEET #	i	ii	iii	iv	v	vi		
ISSUE	-	-	-	-	-	-		
SHEET #	1	2	3	4	5	6	7	8
ISSUE	-	-	-	-	-	-	-	-
SHEET #	9	10	11	12	13	14	15	16
ISSUE	-	-	-	-	-	-	-	-
SHEET #	17	18	19	20	21	22	23	24
ISSUE	-	-	-	-	-	-	-	-
SHEET #	25	26	27	28	29	30	31	32
ISSUE	-	-	-	-	-	-	-	-
SHEET #	33	34	35	36	37	38	39	40
ISSUE	-	-	-	-	-	-	-	-
SHEET #	41	42	43	44				
ISSUE	-	-	-	-				

P148505.PDF [Page: 7 of 108]

ISSUE RECORD (continued)

SHEET #	A-1	A-2	A-3	A-4	A-5	A-6	A-7	A-8
ISSUE	-	-	-	-	-	-	-	-
SHEET #	A-9	A-10	A-11	A-12				
ISSUE	-	-	-	-				
SHEET #	B-1	B-2	B-3	B-4	B-5	B-6	B-7	B-8
ISSUE	-	-	-	-	-	-	-	-
SHEET #	B-9	B-10	B-11	B-12	B-13	B-14	B-15	B-16
ISSUE	-	-	-	-	-	-	-	-
SHEET #	C-1	C-2	C-3	C-4	C-5			
ISSUE	-	-	-	-	-			
SHEET #	D-1	D-2	D-3	D-4	D-5	D-6	D-7	D-8
ISSUE	-	-	-	-	-	-	-	-
SHEET #	D-9	D-10	D-11	D-12	D-13	D-14		
ISSUE	-	-	-	-	-	-		
SHEET #	E-1	E-2	E-3	E-4				
ISSUE	-	-	-	-				

P148505.PDF [Page: 8 of 108]

TABLE OF CONTENTS

		PAGE
1.0	**INTRODUCTION**	1
2.0	**ALGORITHM OVERVIEW**	3
2.1	Analysis Packet Formation	3
2.2	Pulse Tagging	3
2.3	Deinterleaving	5
2.4	Fine Channel Monitoring	5
2.4.1	Emitter Activity Monitoring	5
2.4.2	Significant Parameter Change Monitoring	6
2.4.3	Intermittent Scan Monitoring	6
2.4.3.1	Scan Envelope Data Collection	6
2.4.3.2	Scan Mode Change Monitoring	6
2.4.4	Persistent Pulse Stream Monitoring	6
2.4.5	Fine Channelization Performance	6
2.5	Alternate Algorithm Architecture	7
3.0	**FINE CHANNELIZER TOP-LEVEL HARDWARE ARCHITECTURE**	9
3.1	Proposed Software Architecture	9
3.2	Alternate Event Queuing	13
3.3	Top-Level Concurrency	13
4.0	**TAGGER PROCESSING ARCHITECTURE**	14
4.1	Tagger Control Architecture	14
4.1.1	Stream Buffer Input	14
4.1.2	Pulse Tagging Operation	14
4.1.3	Pulse Extraction	16
4.2	Sequence Tagger Architecture	17
4.2.1	Instance Variables	17
4.2.2	Stream Buffer Access	20
4.2.3	Interval Computation	20
5.0	**SMALLTALK SEQUENCE TAGGER MODEL**	22
5.1	VHDL Sequence Tagger Model	22
5.1.1	Circuit Description	22
5.1.2	VHDL Tagger Performance	25
5.2	Experimental Results	26

P148505.PDF [Page: 9 of 108]

TABLE OF CONTENTS (continued)

		PAGE
5.3	Smalltalk Model	26
6.0	**CONCLUSION**	39
6.1	Architecture Selection	39
6.2	Prototype Development	41
6.2.1	Development Plan	41
7.0	**REFERENCES**	44

LIST OF FIGURES

2-1	Fine Channelization Functional Architecture	4
2-2	Fine Channelization Modified Architecture	8
3-1	Fine Channelizer Top-Level Architecture	10
3-2	Fine Channelizer Software Architecture	11
4-1	Tagger Control Architecture	15
4-2	Sequence Tagger Architecture	18
5-1	Tagger Model: Top-Level Block Diagram	23
5-2	Simulation Timing Results	27
5-3	Simulation Timing Results	29
5-4	Simulation Timing Results	30
5-5	Emitter E015	34
5-6	Emitter E069 -- Single Interval Tagger	35
5-7	Emitter E104 -- Single Interval Tagger	36
5-8	Emitter E126	37
6-1	Fine Channelization	40
6-2	Fine Channelizer H/W Block Diagram	42

LIST OF TABLES

5-1	Monitored Sequence Tagger -- State Transition Counts	31
5-2	Measured State Transitions	38

P148505.PDF [Page: 10 of 108]

TABLE OF CONTENTS (continued)

PAGE

APPENDICES

A	StreamBuffer and SequenceStreamTagger (Smalltalk Class Fileout)	A-1
B	MonitoredStreamBuffer (Smalltalk Class Fileout)	B-1
C	Smalltalk State Machine Model Experimental Results	C-1
D	VHDL Source Code Listings	D-1
E	Mentor Quicksim II Start-up Shell Script	E-1

P148505.PDF [Page: 11 of 108]

1.0 INTRODUCTION

During the development of the CANEWS 2 ESM processor (AMEP), an emitter pulse data sorting technique, called channelization, was developed to facilitate the analysis of individual emitters in the Electromagnetic Environment (EME). This channelization process involves the separation of time interleaved pulse data obtained from a set of radar ESM receivers. Channelization occurs in two stages; coarse channelization followed by fine channelization. Coarse channelization refers to the process of separating the pulse environment into separate parallel channels or streams of data based on clustering in the RF, bearing, and pulse width parameter space.

New coarse channels are created spontaneously, using algorithms which are partially implemented in hardware to detect unique clusters of pulses in the three dimensional environment. Once a coarse channel is established, the corresponding pulse data is analyzed to determine the PRI and fine RF characteristics of the emitter. When a successful determination of these values has been achieved, a fine channel is created containing algorithms that remove pulses from the coarse stream that match the known PRI and fine RF pattern. The remaining pulses in the stream are therefore unique and must belong to a new emitter or emitter mode.

Apart from monitoring the residual data from the fine channelization process, several other characteristics of the emitter signal are monitored. The entire list is as follows:

(a) **pri mode changes**: causes residual data during the fine channelization process;

(b) **emitter activity changes**: changes in the state of the emitter (transmission ceased or resume transmission);

(c) **significant changes in bearing and pulse width**;

(d) **intermittent scans**: scan envelope data collection and scan mode changes in period or pattern (only circular or sector scans are currently monitored);

(e) **persistent pulse stream**: the first indication that an emitter may be locked-on; and

(f) **fine channelization performance**: the quality of performance of the fine channelization process during the initial set-up stage.

In the current version of AMEP, the above mentioned fine channelization and monitoring functions are performed in software within the channelAdministrator actor in the preprocessor. The real-time performance throughput of this Smalltalk code has been measured to be about 150 pulses per second. Since the expected pulse density of non-trivial environments can range from tens of thousands of pulses per second to hundreds of thousands of pulses per second, at least two orders of magnitude improvement in real-time performance of these functions will be required.

In order to meet the above-mentioned real-time performance, this study will address the optimization of these algorithms in special purpose hardware. The approach taken will

P148505.PDF [Page: 12 of 108]

1.0 **INTRODUCTION** (continued)

involve the development of VHSIC Hardware Description Language (VHDL) models of specific elements of the channelization processing. This will allow the feasibility of implementing these algorithms in hardware using Field Programmable Gate Array (FPGA) technology to be determined as well as real-time performance of the hardware to be measured. The use of programmable devices in future prototype development would permit modification of the algorithms as the AMEP software detail design progresses. If performance limitations or component density considerations dictate, the design can easily be targeted to an Application Specific IC (ASIC) because the model has been developed using VHDL.

P148505.PDF [Page: 13 of 108]

2.0 ALGORITHM OVERVIEW

The basic functional architecture for the fine channelization process is shown in Figure 2-1. The functions for this process are applied to the pulse streams from each of the coarse channels separately. Detailed descriptions for the algorithms mentioned herein can be found in the AMEP Software Requirements Specification [1] and the AMEP Algorithm Manual [2].

2.1 ANALYSIS PACKET FORMATION

Pulses from a coarse channel pulse stream are grouped to form a pulse packet by the analysis packet function. This packet is formed if any of the following packetization rules become true:

(a) **time gap**: if a time gap exceeding a threshold is detected in the pulse stream, the pulse group preceding that gap forms a packet;

(b) **pulse count**: if the pulse count within the current packet being built exceeds a threshold, the pulse group preceding this condition forms a packet; or

(c) **packet duration**: if the time of the last pulse in the packet minus the time of the current pulse in the stream exceeds a threshold, then the preceding pulses that were put into the packet form a completed packet.

Once an analysis packet is built, the follow-on processing functions are invoked.

2.2 PULSE TAGGING

The first of these functions is involved with the determination of the fine channels. Fine channels created for each known emitter in a coarse channel are based on the results of pulse train analysis performed in the AMEP Track actor. Pulses in the channel stream are identified for each fine channel on the basis of these known PRI and fine RF patterns. The process of selecting these pulses is performed by the pulse tagging function. A pulse tagger is created for each known emitter in the coarse channel. Each tagger is provided with PRI pattern and interval limit values as well as the fine RF characteristics for the known emitter.

Once the above-mentioned analysis packet has been created, each tagger is directed to analyze the pulses in that packet and mark any which match the PRI interval(s) and fine RF characteristics defined for that tagger. In the current AMEP design, any one of the following three types of PRI taggers could be invoked:

(a) **multi-interval tagger**: can match on any one of a set of programmed intervals, no dropped pulse capability;

(b) **single-interval tagger**: can match on only one interval, dropped pulse capability included; or

P148505.PDF [Page: 14 of 108]

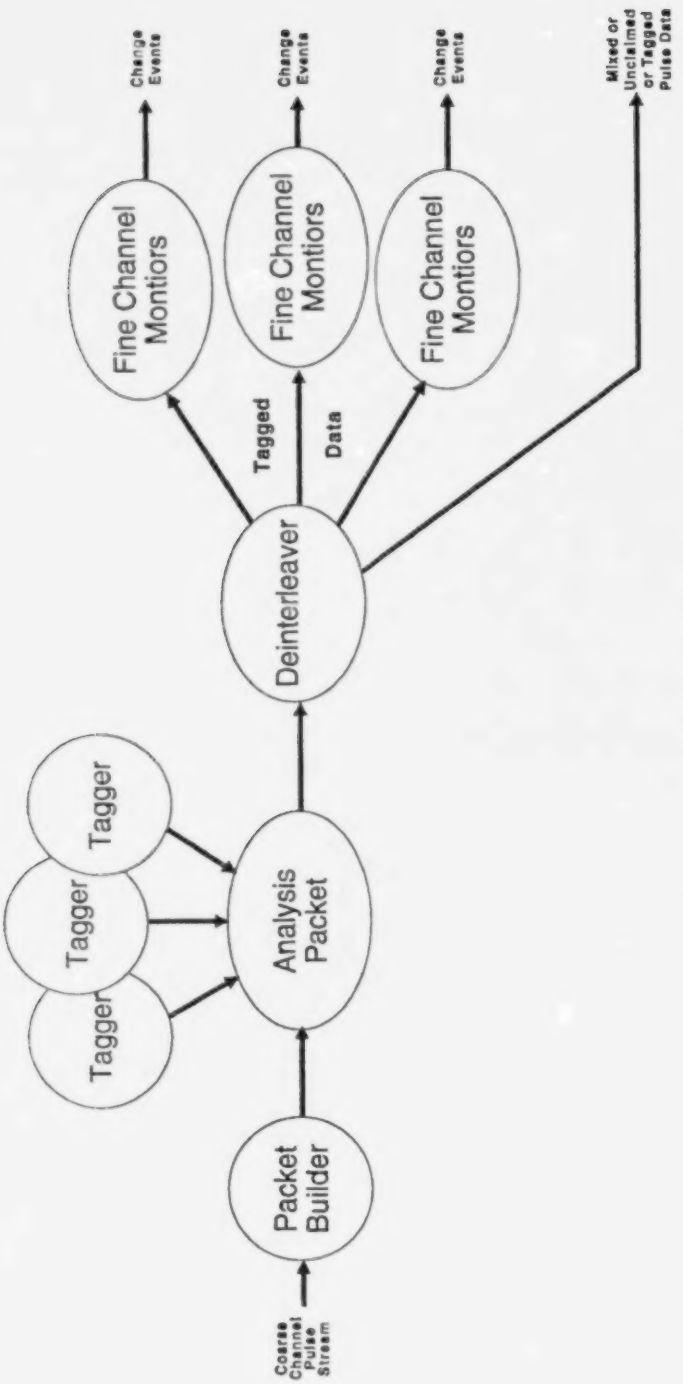


Figure 2-1 CHANNELIZATION FUNCTIONAL ARCHITECTURE

P148505.PDF [Page: 15 of 108]

2.2 PULSE TAGGING (continued)

(c) **sequence tagger**: can match on an ordered sequence of PRI intervals, dropped pulse capability included.

The multi-interval tagger is invoked only when the PRI sequence has not been determined. The sequence tagger is used when the PRI sequence and intervals are known. Although the sequence tagger can be used for a single interval, the single-interval tagger is slightly more efficient for this case.

2.3 DEINTERLEAVING

Once the pulses within the analysis packet are tagged by the tagger(s), they are deinterleaved by the deinterleaver function into the packets belonging to the fine channels. This deinterleaving process may or may not be possible, based on the number of untagged pulses remaining in the analysis packet as well as the number of contested pulses (ones which are tagged by more than one tagger). If the analysis packet contains too many unclaimed or contested pulses or if it is entirely unclaimed, the analysis packet is declared as a mixed or completely unclaimed and is sent on to the follow-on AMEP processing algorithms for further analysis.

If the above-mentioned anomalous cases have not been detected, the analysis packet becomes a fine channel packet if it contains pulses tagged by only one tagger; otherwise it is deinterleaved into more than one packet, each containing pulses tagged by each tagger involved. If processing in the main AMEP subsystem is in tagging mode for a particular fine channel, these packets are sent on to the Track actor associated with the fine channel. These packets are also sent on to the fine channel monitoring functions which are described below.

2.4 FINE CHANNEL MONITORING

The purpose of fine channel monitoring is to detect changes in the state and mode of the known emitters as well as to collect scan envelope information. If changes are detected, the channelAdministrator actor informs the AMEP follow-on processing of the changed parameters or states. The following monitoring functions are enabled once a fine channel is established. The details of these algorithms can be found in the AMEP Algorithm Manual [2].

2.4.1 Emitter Activity Monitoring

The emitter activity monitor detects when the emitter falling in to a fine channel ceases transmission or when it resumes transmission. Upon either event, a message is sent from the channelAdministrator to the main AMEP processing subsystem indicating the event type and fine channel number.

P148505.PDF [Page: 16 of 108]

2.4.2 **Significant Parameter Change Monitoring**

This monitor maintains a history of means and spans (value ranges) for bearing and pulse width. If either parameter deviates from the history, an event is sent from the channelAdministrator to the main AMEP processing subsystem indicating the event type, the new parameter mean and span as well as the associated fine channel number.

2.4.3 **Intermittent Scan Monitoring**

The function of intermittent scan monitoring in the preprocessor is to collect scan envelope data and to detect changes in the scan mode of emitters in a fine channel.

2.4.3.1 **Scan Envelope Data Collection**

When a fine channel is established, a scan analysis is performed by the main AMEP processor. The data for this analysis is requested from the channelAdministrator. This monitor collects envelopes of amplitude averaged samples, each having a duration of approximately eight seconds and sends these envelopes to the scan analysis functions. Once a successful scan analysis has been accomplished. When this occurs, this envelope collection function is terminated.

2.4.3.2 **Scan Mode Change Monitoring**

This monitor is capable of monitoring a circular or sector scan type. It is enabled once the scan type and period(s) are determined. If either the type or period changes significantly, the channelAdministrator sends a message to the main AMEP processing subsystem indicating the event type and fine channel number. This may cause a scan re-analysis or may simply result in the termination of scan monitoring.

2.4.4 **Persistent Pulse Stream Monitoring**

The persistent pulse stream monitor detects the condition where the emitter is constantly illuminating the own ship. This can occur when the emitter is lock-on to own ship or is close in range from the own ship, resulting in the reception of side lobe signals. When this condition is detected and subsequently if an intermittent signal condition is detected, the channel administrator will send a message to the main AMEP processing subsystem indicating the condition detected, a scan envelope of averaged amplitude information for the last one second and the fine channel number in which the condition occurred.

2.4.5 **Fine Channelization Performance**

When a fine channel is initially established, it exists on a trial basis. This monitor determines the percent of the pulse train which is tagged. A one time only message is then sent to the main AMEP processing subsystem indicating this percentage as well as the fine channel number. If too many untagged pulses occur, the tagger corresponding to this fine channel will be re-tuned following further data analysis by the Track subsystem within the main AMEP processor.

P148505.PDF [Page: 17 of 108]

2.5 ALTERNATE ALGORITHM ARCHITECTURE

The main disadvantage of the above architecture is that the taggers are forced into acquisition mode (they must lose lock) at the beginning of each new packet of data in each coarse channel. In some cases, this may cause packets with few pulses to remain untagged since the tagger cannot regain lock-on fragmented data. It is therefore more desirable for a tagger to remain locked on to a pulse train without regard for packet boundaries (see the section on Analysis Packet Formation above). The architecture for this process, called stream tagging, is shown in Figure 2-2 and the Smalltalk version of the SequenceStreamTagger and associated StreamBuffer is listed in Appendix A.

In this new architecture, the coarse channel pulse stream is temporarily held in a stream buffer so that the taggers can match on new data. The taggers begin in acquisition mode and ultimately lock-on to the stream of pulses. As the stream is filled from the input end, the taggers tag and relinquish pulses within the stream towards the output end. The taggers only lose lock if the number of dropped pulses in the stream exceed their respective thresholds.

The deinterleaver, extracts relinquished pulses from the output of the stream buffer. These are deinterleaved on the basis of tags as described in a previous section. The resulting fine channel streams are input to the monitors.

The deinterleaved streams are also packetized by the packet builder so that analysis within the main AMEP processor subsystem can be triggered in a timely manner.

P148505.PDF [Page: 18 of 108]

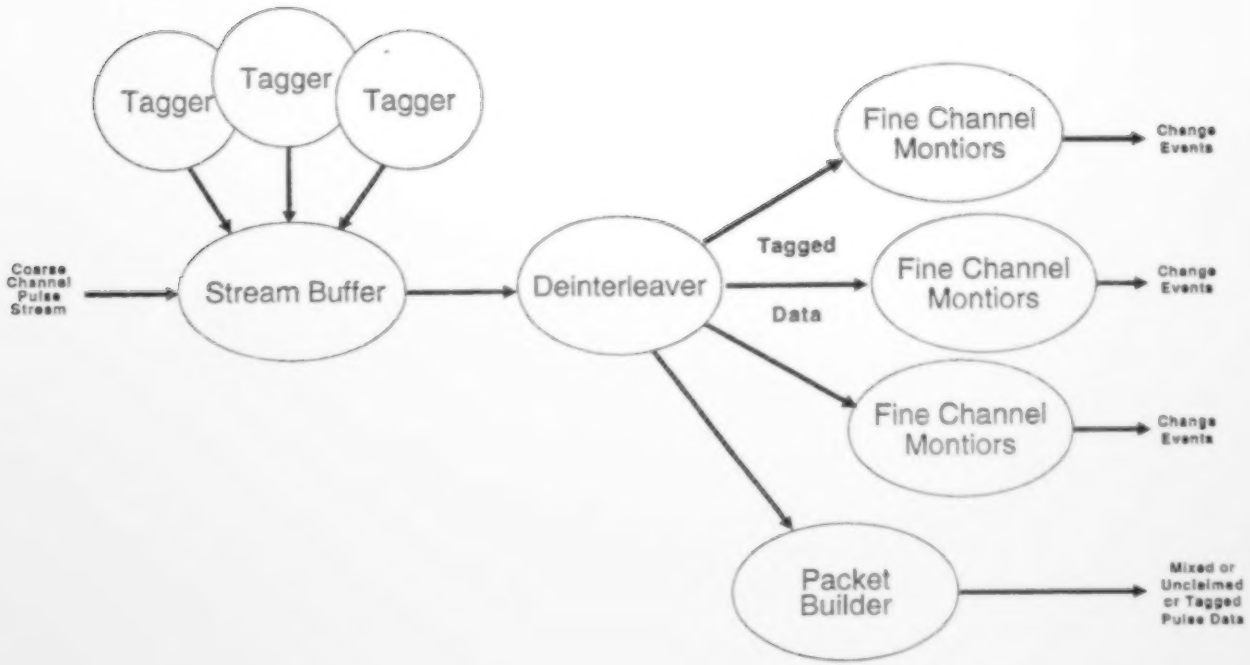


Figure 2-2 FINE CHANNELIZATION MODIFIED ARCHITECTURE

P148505.PDF [Page: 19 of 108]

3.0 FINE CHANNELIZER TOP-LEVEL HARDWARE ARCHITECTURE

For each coarse channel, the algorithms described above participate in the formation of one or more fine channels as well as monitor changes in the fine channel pulse stream data. Considering the inadequate real-time performance of the Smalltalk implementation of these algorithms, it has been proposed that all of the pulse processing in the preprocessor be implemented in hardware. A possible top-level architecture for this implementation is shown in Figure 3-1.

For a single coarse channel, the pulse stream is applied to the stream buffer. Each tagger operates on the pulses in the stream, computing inter-pulse time differences. These differences are determined using the arrival time of a reference pulse and the arrival time of a more recent pulse in the stream. As each tagger algorithm searches for their corresponding PRI intervals, several pulses in the stream may be skipped in order to match on a correct interval.

When a tagger finds the correct interval, the more recent pulse is tagged and the reference becomes the newly tagged pulse. In this manner, pulses become de-referenced by the taggers as they move forward to consider more recent pulse train fragments. The function of the extractor is to remove pulses from the stream which have been de-referenced by all taggers. The extractor transmits these pulses on the internal pulse bus to the output pulse buffers and the monitors as shown in Figure 3-1.

The output pulse buffers are circular queues which overwrite when they become filled. Circular queues insure that the most recent information can be extracted under degraded modes of operation due to high pulse data rates in a channel. There is a separate pulse buffer for each fine channel and a single buffer which is used to capture unclaimed data for each coarse channel.

Fine channel pulse data is also applied to a set of monitors associated with each tagger. Events such as transmission ceased, an intermittent scan envelope, etc. are queued in the event buffers as shown in the figure. These queues are FIFO in nature and correspond to each fine channel in existence. In order to monitor data from a new emitter mode, an unclaimed monitor channel is provided. A corresponding unclaimed event buffer is used to queue these events for each coarse channel.

A single event buffer is provided for high priority events (ones which must be serviced immediately by the software). This single buffer is fed by the high priority monitors in all coarse channels.

3.1 PROPOSED SOFTWARE ARCHITECTURE

The proposed software architecture that would process data and events from the fine channels is shown in Figure 3-2. In the present AMEP software, a Track is established for each emitter mode. The associated fine channel sends data to that Track in the form of raw pulses and monitored events. Once the Track has classified the signal based on the raw pulse stream, reading of the raw pulse buffer ceases. The events, however, continue to flow from the fine channel into the Track.

P148505.PDF [Page: 20 of 108]

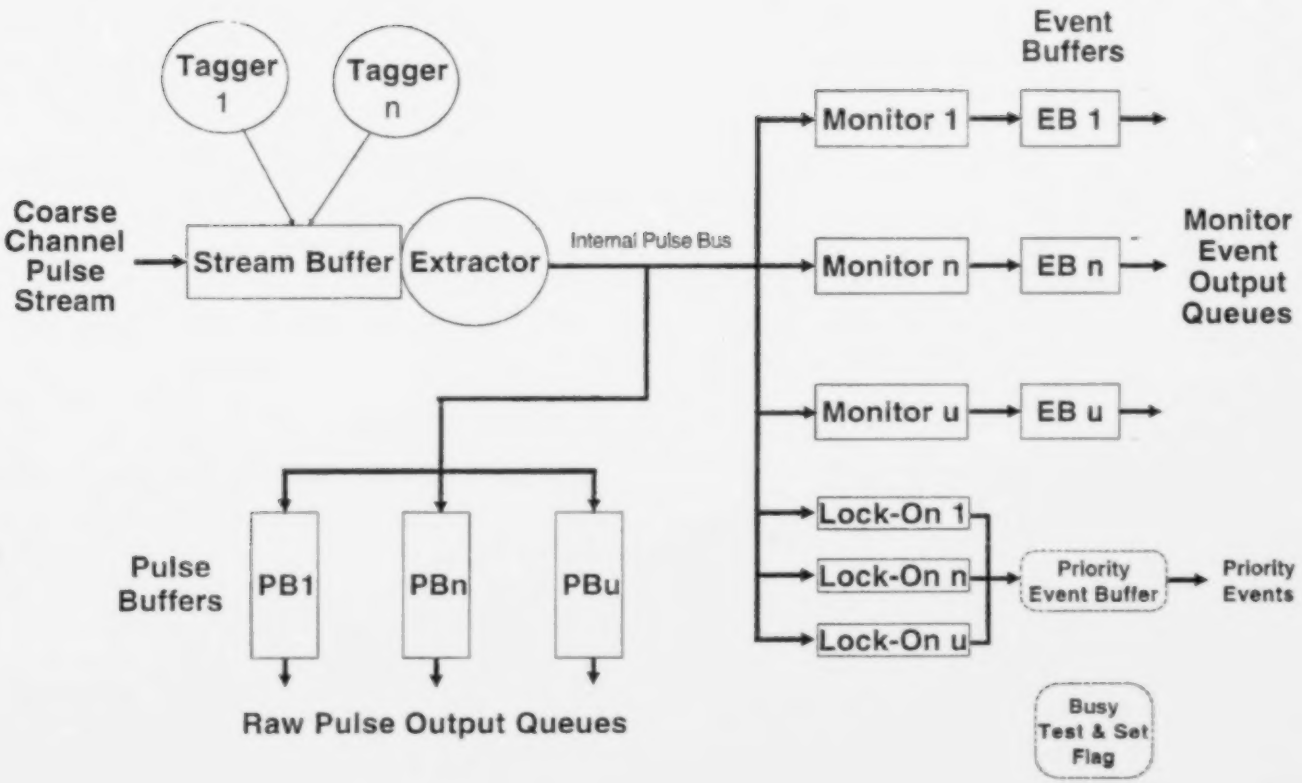


Figure 3-1 FINE CHANNELIZER TOP-LEVEL ARCHITECTURE

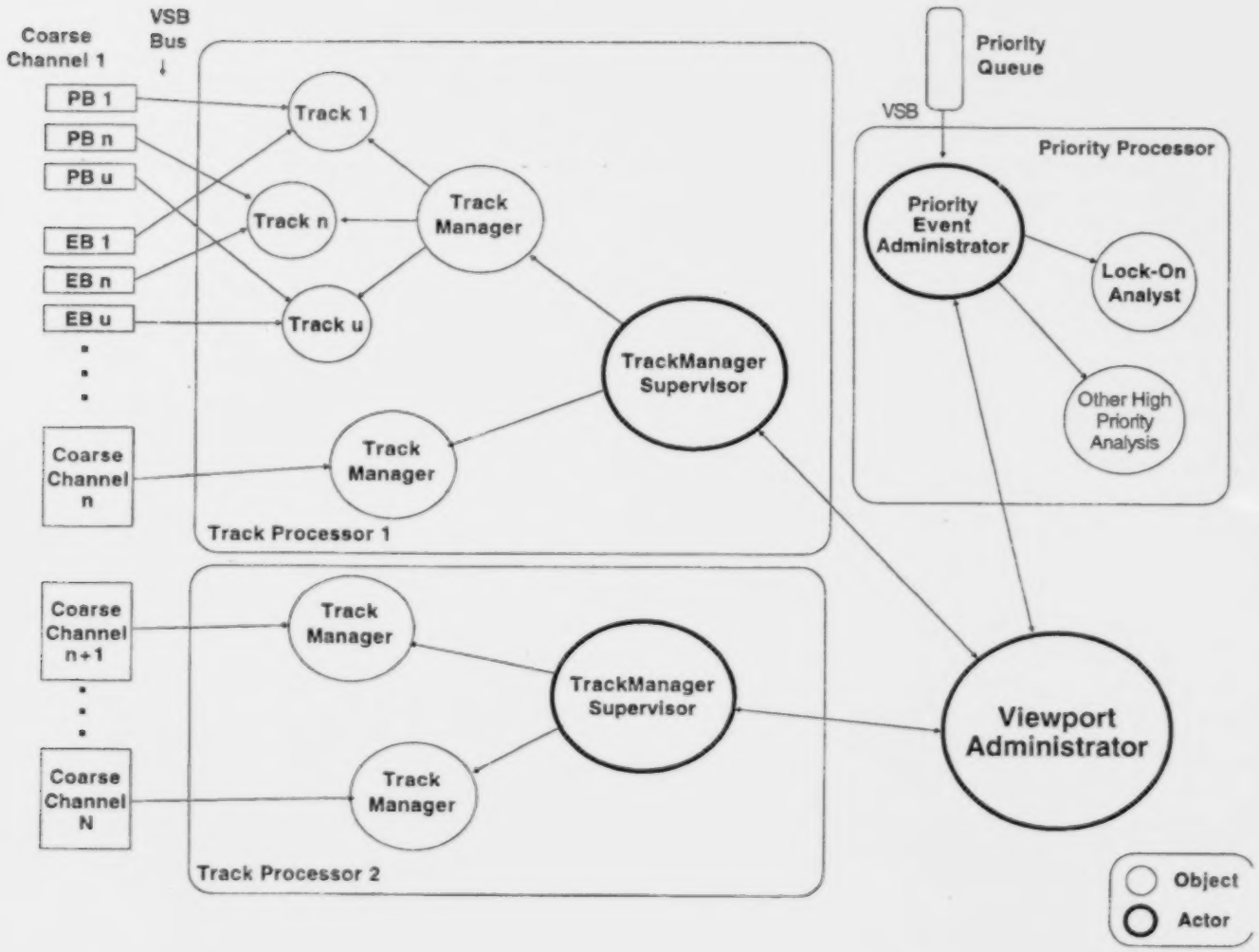


Figure 3-2 FINE CHANNELIZER SOFTWARE ARCHITECTURE

P148505.PDF [Page: 22 of 108]

3.1 PROPOSED SOFTWARE ARCHITECTURE (continued)

The architecture shown in the figure facilitates processing by multiple Tracks on each of multiple processors. Track processing is ultimately managed on each processor by instances of the TrackManagerSupervisor (TMS). These supervisors hold a collection of TrackManagers (TMs), each of which encapsulates one or more Tracks associated with a single coarse channel. The TMS directs each TM to execute a processing cycle in a round robin manner. Between each TM processing cycle, the TMS invokes a tryReceive, to detect and service any incoming messages.

Each TrackManager, in turn, directs its Tracks to perform a processing cycle. These low level processing cycles involve dequeing a bounded number of pulses from the associated pulse buffer and a single event from the associated event queue. After performing one or more processing cycles, the Track may forward a TrackReport to the TMS via the associated TM. This report is then sent out to the ViewportManager to be distributed to other AMEP actors.

Although the Track processing mentioned above is somewhat bounded in execution time, from previous performance measurements, a single Track could consume several seconds of time on any one cycle. Since preemptive scheduling is not available in Actra (the multi-tasking operating system used in AMEP), no high priority events can begin to be processed during these Track processing cycles. Therefore, a separate high-priority mechanism is required in order to provide a timely response to priority monitor events from the preprocessor. This mechanism is centred around the PriorityEventAdministrator which polls the single priority event queue from the hardware. Events such as lock-on detect can then be processed on the separate processor without being affected by long Track processing delays.

Other processors could be used to increase the response time to priority events, where each would have instances of worker actors which would carry out concurrent analysis associated with the events. The results from the priority analyses are sent to the ViewportManager via the PriorityEventAdministrator.

Algorithm tuning could occur directly by the TrackManager or Track itself as required. In order to prevent interleaved accesses to the hardware processor, the software residing on each processor would have to be provided with a common test and set flag, most likely residing on the pulse processor. Any algorithm tuning process would then loop on the flag indicating that the processor is busy. When free, the tuning process would resume to completion. This process is required because tuning of the algorithms involves several data transfers across the VSB bus, where the number of data words is variable depending on the tuning parameter. While the tuning is in progress, the hardware must not be allowed to process any pulses.

Load balancing among the Track Processors is an issue that will be addressed during future AMEP software development.

P148505.PDF [Page: 23 of 108]

3.2 ALTERNATE EVENT QUEUING

Events from the different monitor types within a fine channel are described by differing amounts of data. In the case of the intermittent scan envelope, the amount of data within the envelope event is variable. In order to minimize the requirement to support variable record sizes within a single output queue, it has been proposed that a queue for each monitor type be implemented for each fine channel. This would reduce the requirement to support variable record sizes to only one queue.

The chronological order of events within the queues of Figure 3-1 is important with respect to Track processing. In order to accommodate event queues for each monitor type, the a non-destructive read capability will have to be provided for each queue so that the event time can be read in software. This capability would facilitate the ordering of events in a fine channel.

3.3 TOP-LEVEL CONCURRENCY

Concurrency at the top level can be exploited in order to increase the real-time performance of the pulse processing part of the preprocessor. The highest level of concurrency could be structured at the coarse channel level, resulting in the parallel execution of the fine channelization and monitoring algorithm set for each coarse channel. Although this architecture would be somewhat ideal in terms of real-time performance (there can be up to about 200 coarse channels that could execute in parallel), the replication of the hardware real estate required for these algorithms by a factor of about 200 would likely be prohibitive. This is due to the expected complexity of the tagging and monitoring algorithms.

A hybrid approach to the above technique could be to allocate a number of coarse channels to a single pulse processor which would perform the tagging and monitoring algorithms. The real-time improvement would then be approximately related to a practical number of pulse processors. Depending on the level of complexity of the algorithm implementations, a single instantiation of the pulse processor may only be practical.

Since the amount of hardware real estate required is the subject of this study, it will not be possible to estimate the degree of concurrency that could be achieved for the hybrid approach at this time.

Concurrency can also be achieved within an instantiation of the pulse processor by the parallel execution of each of the taggers and each of the monitors. This pipe-lining approach would be triggered upon the input of each pulse into the processor. Since the hybrid processing technique described above seems more practical, the coarse and fine channel numbers associated with the pulse would have to be latched internally throughout the pipeline.

For the purposes of this study, the internal pipe-lining approach will be assumed. The detailed architecture of the tagger subsystem including the stream buffer and extractor is discussed in the following section.

P148505.PDF [Page: 24 of 108]

4.0 TAGGER PROCESSING ARCHITECTURE

The processing architecture for the tagger and associated stream buffer has been divided into to layers, namely the tagger control architecture and the architecture of the tagger algorithm itself. These will be described in the following two subsections.

4.1 TAGGER CONTROL ARCHITECTURE

The tagger control architecture is shown in Figure 4-1. The main components at this level of abstraction are the stream buffer and associated access control functions, the actual sequence tagger and a system controller (channel selector). The channel selector manages the tagging and extraction processing functions for each enabled fine channel.

From a top-level point of view, pulse input to the stream buffer, pulse extraction and the tagging process occur concurrently. These three operations are described in the following subsections.

4.1.1 Stream Buffer Input

Pulses associated with a particular coarse channel are input to the channel's stream buffer by the loader. The loader is a state machine which is activated when the coarse channelization hardware has completed a filtering and mapping function for a pulse. At this time, the coarse channel number and associated pulse descriptor is available for storage into the corresponding coarse channel stream buffer.

When the loader is activated, it obtains access to the stream buffer with permission from the array access arbiter. When access has been acknowledged, the PDW is appended to the stream buffer addressed by the coarse channel number. The stream buffer acts as a FIFO; thus, if the buffer is full, the PDW is discarded. When the pulse storage operation is complete -- including the update of the write pointer and FIFO pulse count for the associated coarse channel, access to the buffer is released. The arbiter then resumes polling its clients, searching for other access requests.

4.1.2 Pulse Tagging Operation

While pulses are being input to the stream buffers in any undetermined order, the sequence tagger operates on pulse data within stream buffers according to a scheduling algorithm. This concurrent operation is scheduled so that a bounded number of pulses from each enabled fine channel are processed at a time. This form of scheduling has been selected so that high data rate channels do not capture the available processing time. For the high rate channels, if tagging and subsequent pulse extraction falls behind, data will be lost at the input to the FIFO as described in the preceding subsection.

The above-mentioned scheduling operation is conducted by the channel selector. When a tagger is created, the associated fine and coarse channel numbers are stored into an active channel list by the AMEP multiprocessor. The channel selector then scans this list, issuing start processing cycle requests to the sequence tagger along with the fine and coarse channel number. This number is used as the most significant address bits of memories

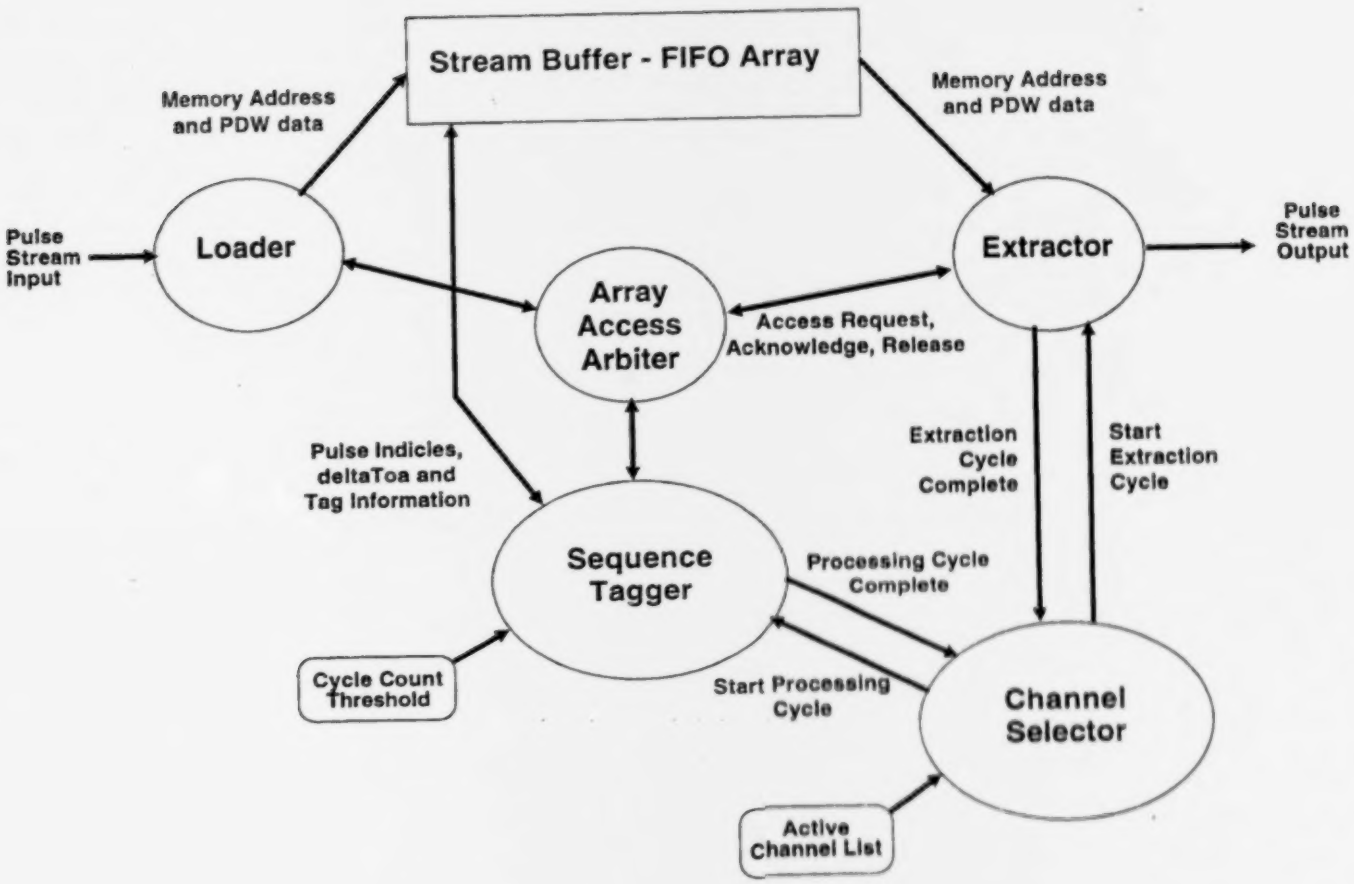


Figure 4-1 TAGGER CONTROL ARCHITECTURE

P148505.PDF [Page: 26 of 108]

4.1.2 **Pulse Tagging Operation** (continued)

containing the specific state for each tagger instance (selection of the tagger instance will be more fully described later in this report).

Upon reception of the above-mentioned message, the sequence tagger executes processing cycles until either no more pulse data is found in the corresponding stream buffer or until the number of cycles exceed a cycle count threshold (the cycle count threshold is used to place an upper bound on the processing time-slice for each fine channel). When the processing cycle is terminated, the tagger sends the processing cycle complete message to the channel selector. The selector then obtains the next fine channel from the list and repeats the above process.

While processing cycles are performed by the tagger, occasional access to the stream buffer is required. These accesses include fetching the TOA of pairs of pulses in the stream, determining when no more data is available and appending a tag (fine channel number) to pulses that match expected PRI patterns. When these accesses are required, the tagging state machines request access to the stream buffer via the array access arbiter. When access is obtained, the required operation(s) are performed, followed by a release response to the arbiter. Note that is expected that access to the buffer occurs for a relatively low proportion of the tagging algorithm execution, thus, the loading and extraction functions associated with the buffer can occur virtually concurrently.

A more detailed examination of the sequence tagger architecture is presented in a following section of this report.

4.1.3 **Pulse Extraction**

Pulse extraction can only occur if pulses at the end of each stream buffer are unreferenced (e.g., all taggers in that channel have completed their consideration of the pulse stream using those pulses). Unreferenced pulses, or pulses which have remained referenced for times exceeding a time-out threshold are removed by the extractor and would be sent on to the monitoring functions described earlier in this report. These pulses are also stored in buffers specific to each fine channel for access by the AMEP processor.

The extraction process also operates in a round robin manner, scheduled by the channel selector. For each coarse channel in the active channel list, the channel selector sends the start extraction cycle to the extractor along with the coarse channel number.

The extraction cycle consists of obtaining access to the stream buffer via the array access arbiter, testing the last pulse in the buffer for the selected channel and transferring that pulse to the output pulse stream if the above transfer criteria are met. This cycle continues until no more data can be transferred. The extraction cycle complete message is then sent to the channel selector so that an extraction cycle can occur for the next active channel.

Since the stream buffer access is released while a pulse is presented on the output bus, access by the loader or sequence tagger can be interleaved with this process.

P148505.PDF [Page: 27 of 108]

4.2 SEQUENCE TAGGER ARCHITECTURE

The sequence tagger code in Appendix A is a Smalltalk class. Each fine channel is associated with an instance of this class. Each instance shares the code provided by the class, however, the state of each instance is represented in a separate copy of the instance variables associated with the class.

This object-oriented approach can be preserved by containing each instance variable in a memory array. In the case of the sequence tagger, the instance is associated with the fine channel number. Operationally, once the fine channel number is defined, the code which is represented by state machines, is directed to execute using the set of instance variables addressed by this fine channel number.

The concept is illustrated in Figure 4-2 where the sequence tagger is shown to consist of four major state machines:

(a) Sync: determines the phase of the sequence that matches the current PRI interval;

(b) MatchLock: achieves PRI sequence lock-on if the required number of successful PRI interval matches are found;

(c) LockOn: tags pulses which match the known PRI sequence; and

(d) SkipPulse: tags a pulse which is separated by a time gap.

These state machines use several instance variables in order to preserve the state and position in the stream buffer once a processing cycle is terminated.

4.2.1 Instance Variables

The instance variables used by the sequence tagger are:

(a) rightLimit: an ordered list of the upper bounds of the known PRI interval elements;

(b) leftLimit: an ordered list of the lower bounds of the known PRI interval elements;

(c) maxT: the largest of the interval limits in the PRI sequence;

(d) minMatchCount: the minimum number of successive matches of the elements in PRI sequence in order to move from the acquisition to the lock-on state;

(e) missingPulseThreshold: the maximum number of allowable missing pulses in the stream before lock-on is lost;

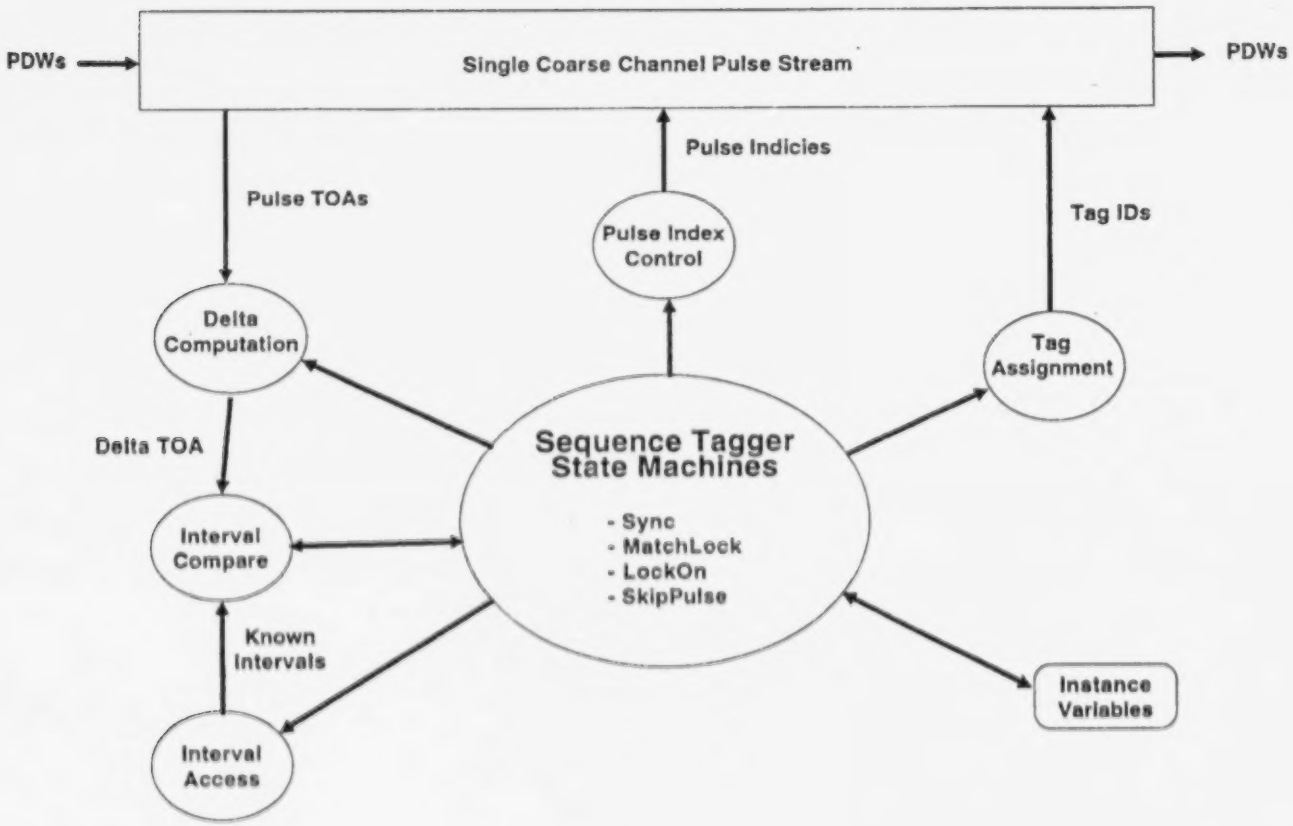


Figure 4-2 SEQUENCE TAGGER ARCHITECTURE

4.2.1 **Instance Variables** (continued)

(f)	numIntervals:	number of PRI intervals in the known sequence;
(g)	taggerType:	the tagger type, currently the type can be NONE or SEQUENCE;
(h)	state:	the major state in which the tagger is currently operating, one of: sync, matchLock, LockOn, or SkipPulse;
(i)	leftAccumulator:	used to store the accumulated left intervals when missing pulses are encountered;
(j)	rightAccumulator:	used to store the accumulated right intervals when missing pulses are encountered;
(k)	phase:	the current phase of the known sequence that has successfully matched the pulse train;
(l)	referenceIndex:	the position in the stream buffer of the reference pulse;
(m)	currentIndex:	the position in the stream buffer of the most recent pulse which is being considered by the algorithm;
(n)	matchedPulses:	a list of pulse indices which correspond to successful matches while the algorithm is in acquisition mode (these pulses will be tagged if lock-on can be achieved);
(o)	matchCount:	the number of successful matches during the acquisition mode; and
(p)	skipCount:	the number of skipped intervals when processing a missing pulse segment while in lock-on mode.

A few of these variables represent lists where elements of the lists are accessed by the algorithm. Since an upper bound can be placed on these lists, the maximum list length is allocated to each instance. Thus, the instance is define by the most significant address bits of these memories while the list element is accessed by the least significant address bits. The algorithm state machines have access to these least significant bits so that the lists can be traversed.

As a consequence of limiting these list lengths as well as consideration of the bit widths of the memories which will represent the single value instance variables, the proposed limitations on the sequence tagger are:

P148505.PDF [Page: 30 of 108]

4.2.1 **Instance Variables** (continued)

(a) maximum stagger level: 16;
(b) maximum number of taggers in a coarse channel: 8; and
(c) maximum number of tagger types: 8.

These limitations can be extended at the expense of employing slightly wider bit fields in the architecture.

4.2.2 **Stream Buffer Access**

As mentioned in a previous section, access to the stream buffer is controlled by the array access arbiter. Once the tagger algorithm establishes access control, several different specialized functions are invoked which use or modify data associated with the pulse stream or the actual pulses themselves.

One of the main functions of the tagging algorithm is to determine the time difference between any two pulses in the stream. These pulses are specified by index number and are selected using the pulse index control function. In the actual hardware, two out of three possible index values are selected by a multiplexer, resulting in the application of the selected pair of index numbers to the stream buffer. Within the state machine logic of the stream buffer, these two indices result in the access of the TOA values associated with the two pulses. These values are used in the interval computation described in the next subsection.

Once a pulse is matched against a tagger's known sequence, a tag id must be written into a field associated with the pulse. This is accomplished by the tag assignment function. This function selects one of the three possible index values available from the tagger and requests that a bit representing the current fine channel number be written into the tag bit field associated with the selected pulse. State machines within the stream buffer handle and acknowledge the completion of this action.

4.2.3 **Interval Computation**

A major function of the tagging algorithm is to compare the known sequence to time intervals between pairs of pulses within the stream. Since these pulses may not be adjacent (pulses from more than one emitter may be time-interleaved), they are independently selected based on their indices within the stream. Once the state machines within the pulse stream obtain the required TOAs, the delta computation function determines the time difference.

This delta is then compared with one of the known intervals from the sequence which is obtained using the right and left interval lists in the instance variables. The tagger state machine takes action on the result of this compare, resulting in a one of the selected pulses being tagged or the interval between another two pulses being tried.

Access to the interval list is controlled by the interval access function. This function can obtain a single upper and lower bound on a predicted interval or can add successive intervals if dropped pulses are suspected. These access and addition functions operate

P148505.PDF [Page: 31 of 108]

4.2.3 **Interval Computation** (continued)

within one or two clock cycles resulting in a very fast determination of the predicted interval.

P148505.PDF [Page: 32 of 108]

5.0 SMALLTALK SEQUENCE TAGGER MODEL

In the tagger architecture described in Section 4.0, it is the Sequence Tagger algorithm that constitutes the majority of the pulse processing. In order to make an assessment of the feasibility of implementing these algorithms in hardware, the Sequence Tagger has been modeled and synthesized using VHDL. The VHDL model state timing was then used to construct a Smalltalk model. The performance of the hardware tagger against several emitter data files in the AMEP database was then estimated using this model and the VHDL timing data.

5.1 VHDL SEQUENCE TAGGER MODEL

The processing modeled in this study is represented schematically in Figure 5-1. The major logic elements that comprise the tagger hardware are:

- pulseStream - pulseStream.hdl
- taggerInst - tagger.hdl
- leftPRILimits - reg.hdl
- rightPRILimits - reg.hdl
- rightAcc - alu32.hdl
- leftAcc - alu32.hdl
- deltaRightCompare - le32.hdl
- deltaLeftCompare - ge32.hdl
- deltaMaxTCompare - ge32.hdl
- matchIndexCtr - 74ls191
- matchIndexMem - reg10.hdl
- matchIndexCompare - 74ls95

The model pulseStream block is implemented as a behavioral model only. The tagger block and the alu32 blocks have been implemented at the Register Transfer Level (RTL) model and have been synthesized using generic gates to permit an estimation of complexity. The models for the match functions (le32, and ge32) and the register files (reg and reg10) were generated with the Mentor Autologic Blocks tool which generates RTL (synthesizable) VHDL code.

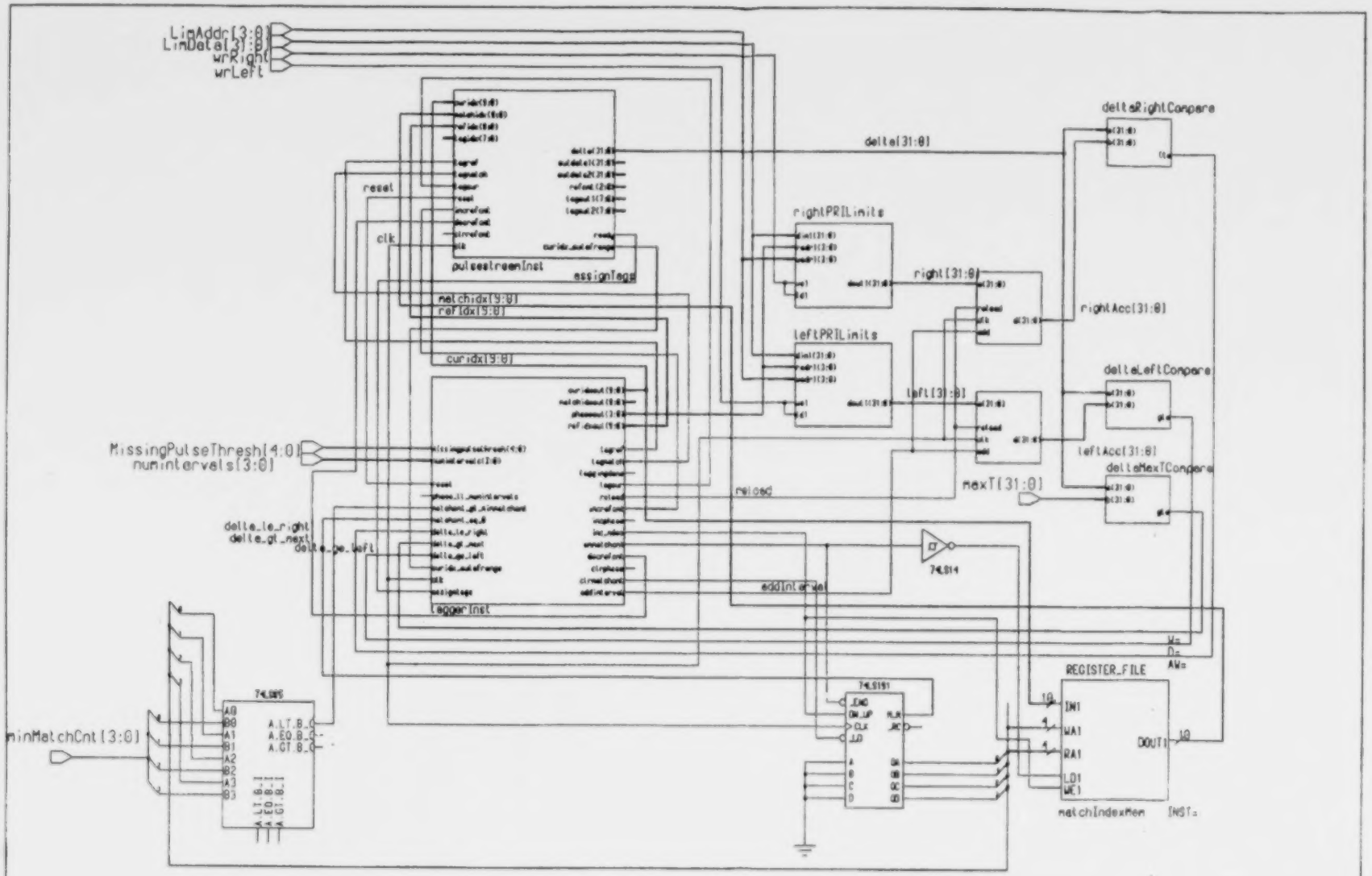
Appendix D includes the source listings for each of the VHDL models for the different logic blocks represented on Figure 5-1.

Appendix E contains the shell script (Mentor "Dofile") that was used to set up the simulator to program the tagger instance variables for processing the test data file.

5.1.1 Circuit Description

PulseStream Block

The pulseStream block encapsulates the stream buffer, the array access arbiter, and the extractor functions represented in Figure 4-1 (tagger control arch). This has been



TAGGER MODEL : TOP LEVEL BLOCK DIAGRAM

TAGGER MODEL: TOP LEVEL BLOCK DIAGRAM Figure 5-1

8505.PDF [Page: 33 of 108]

P148505.PDF [Page: 34 of 108]

5.1.1 **Circuit Description** (continued)

implemented as the behavioral model since it does not contribute significantly to real-time performance or design complexity. The pulseStream also provides access to the pulse data file used for testing the operation of the tagger.

Upon reset, the entire data file (e015.dta) of 1000 PDWs is read in to an internal array of integers. The pulseStream strobes the taggerInst block (Sequence Tagger State Machine) to begin tagging by the assertion of the assignTags signal. This signal remains asserted for the duration of the tagging process.

The pulseStream block also provides the logic for tagging pulses (although not required within the scope of this study) and the 48-bit adder required to calculate the difference between the TOA indexed by the reference and current Index pointers (refIdx and curIdx). This 48-bit value must be rounded off to produce a 32-bit value with μsecond resolution (TOA is stored in 250 nsecond resolution). If the packet builder technique described in Section 2.4 is implemented, the delta TOA value (and associated arithmetic units) could be reduced to 16 - 18 bit data widths since all deltas would be less than the time programmed for the gap rule.

TaggerInst Block

The taggerInst block includes the sequence tagger state machine, instance variables required for skip count processing, pulse indexing into the pulse buffer array, and PRI sequence indexing.

When the assignTags signal is asserted, the tagger initializes all index counters to point to the first pulse in the stream buffer. The tagger algorithm described in detail in Section 3.0 attempts to match PRI values to the programmed limits. Tagging continues until the curIndex signal references beyond the last pulse in the pulse buffer.

Standard high-speed RAM devices can be used to store missingPulseThreshold, numIntervals, minMatchCount, and maxT instance variables required by the tagger. The fine channel number is used to index the specific instance variables for each fine channel. For the purpose of this study, these values are presented as static signals to the tagger state machine.

The selection of a block oriented pulseStream over that of stream-oriented access permits the tagger to operate with fewer memory devices. If the stream-oriented approach was to be implemented in future development work, all instance variables including the phase, pulse index counters, and the accumulated PRI sequences would require temporary storage using standard high-speed static RAM devices.

LeftPRILimits/RightPRILimits Blocks

Register files (generated by Autologic Blocks) are used for storage of the PRI sequence data with a separate file for the low (left) and high (right) PRI limits for each interval in the sequence. The code generated is synthesizable but has not been targeted to any specific logic family.

P148505.PDF [Page: 35 of 108]

5.1.1 Circuit Description (continued)

The block has been defined as a 32-bit wide, 16 location register file, but would likely be implemented as a high-speed static RAM. The sixteen PRI values are adequate for a single tagger, but to support the expected number of fine channels (>200), a 4 k x 32 static RAM will be required. The fine channel number would be used as the high order address lines to select a bank of 16 PRI locations for the fine channel.

The data width of these memory devices would be reduced significantly (16 - 18 bits) if the packet builder technique is implemented in the pulseStream.

LeftAcc/RightAcc Blocks

The 32-bit wide accumulators for the left and right PRI limits are implemented at the register transfer level (RTL) and can be synthesized. These accumulators operate transparently until skipped pulse processing is initiated. The transparent/accumulate operation is determined by the level of the reload signal on the clock edge. When reload is asserted, the PRI limits stored in the register files pass directly through to the output register; when not asserted, the accumulated value is multiplexed to the output.

The width of this register is dependant on the maximum duration over which the tagger can accurately predict pulses; 32 bits has been found to be sufficient.

DeltaLeftCompare/DeltaRightCompare/DeltaMaxTCompare Blocks

The delta compare logic has been generated using the Autologic Blocks VHDL code generator. These logic elements may be implemented in future developments using standard ALU components (16 and 32 bit comparators in sub 100 ns range are commercially available) or this code can be targeted to programmable device technology.

matchIndex Logic Block

A 10-bit wide register file generated using the Autologic Blocks is used in conjunction with a 74 series binary counter for generating the match count index and storage of the match addresses. If the match count exceeds the minMatchCount value for the tagger, the indices stored in the register file are used for tagging. The comparison function is performed by a 74LS85.

5.1.2 VHDL Tagger Performance

The sequence tagger functional performance was manually verified against the tagged pulses produced by the original Smalltalk implementation of the SequenceTagger. The taggers throughput was measured using the Mentor "Quicksim II" and Autologic simulators and for the purpose of this study the tagger was synthesized using the generic (TTL) logic family with unit delays. This provides a simple method of determining the number of state transitions required by the tagger but will not provide technology specific timing analysis. In a prototype system, the minimum clock period for the state machine will be dictated by the propagation delay of the 32-bit ALU (alu32.hdl). Cost/performance/real-estate trade-offs would dictate the optimum implementation for this adder. For simulation

5.1.2 **VHDL Tagger Performance** (continued)

purposes, a 10 MHz clock has been used; this dictates the requirement for sub-100 ns propagation delay for the 32-bit ALU.

Figure 5-2 (i, ii, iii, iv) shows the signals of interest when the tagging process is initiated with the "assignTags" signal asserted. These four graphs illustrate the state transitions required in order to synchronize the PRI sequence in the pulse data train with the PRI sequence programmed for the tagger. The current state machine design requires one clock cycle for tagging once sequence lock-on has been established. Figure 5-2 (iv) demonstrates the match cycle timing. In order to satisfy longer latency times in the 32-bit arithmetic functions, extra wait states have to be added to the "match state" function. This will have minimal impact on overall performance.

Figure 5-3 (i, ii) illustrates the state transition timing of the tagger when the tagger loses synchronization, and subsequently must re-acquire lock-on. At the 6400 ns point in time, the skip count starts increasing until the maximum number of missing pulses is exceeded. The tagger then re-enters the acquisition (assign1) state.

Figure 5-4 illustrates the final cycles required by the sequence tagger to complete tagging of the e015 data file. The time required for the sequence tagger to process 1000 pulse descriptor words was 274,100 ns based on a 10 MHz cycle time. This corresponds to 3.65 million pulses per second for the case of one tagger in a single course channel.

5.2 SMALLTALK MODEL

In order to asses the real-time pulse throughput of the sequence tagger algorithm, the Smalltalk sequence tagger of Appendix A has been restructured into the fine grain functions of the monitored sequence tagger listed in Appendix B.

Table 5-1 lists a total of 31 functions which are performed during the execution of the algorithm. This estimate is based on the actual VHDL hardware simulation described in the next section of this report. In some cases, the number of state transitions have been zero costed. This implies that these operations occur in parallel with other non-zero costed operations and therefore do not contribute any additional execution time.

While executing, the algorithm accumulates a total number of state transitions for each of the 31 functions. These totals can then be represented in histograms to show the relative usage of each function as well as the total number of state transitions which can be used to indicate the pulse throughput of the algorithm. This latter computation is based on a minimum state dwell time allowing for reliable hardware operation.

5.3 EXPERIMENTAL RESULTS

The monitored sequence tagger was run on recorded pulse data from four files.

Emitter File E015 consisted of a four level staggered pulse train which changed PRI modes to a single interval pulse train. Several mode changes can be observed in the first

P148505.PDF [Page: 37 of 108]

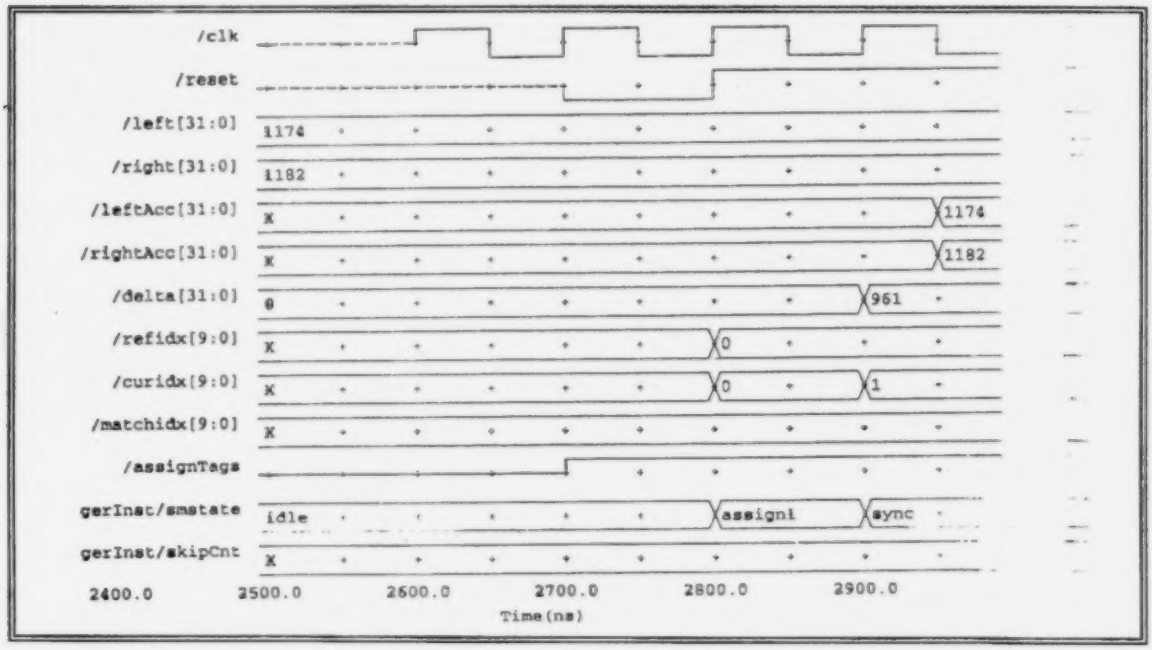


Figure 5-2 (i) SIMULATION TIMING RESULTS

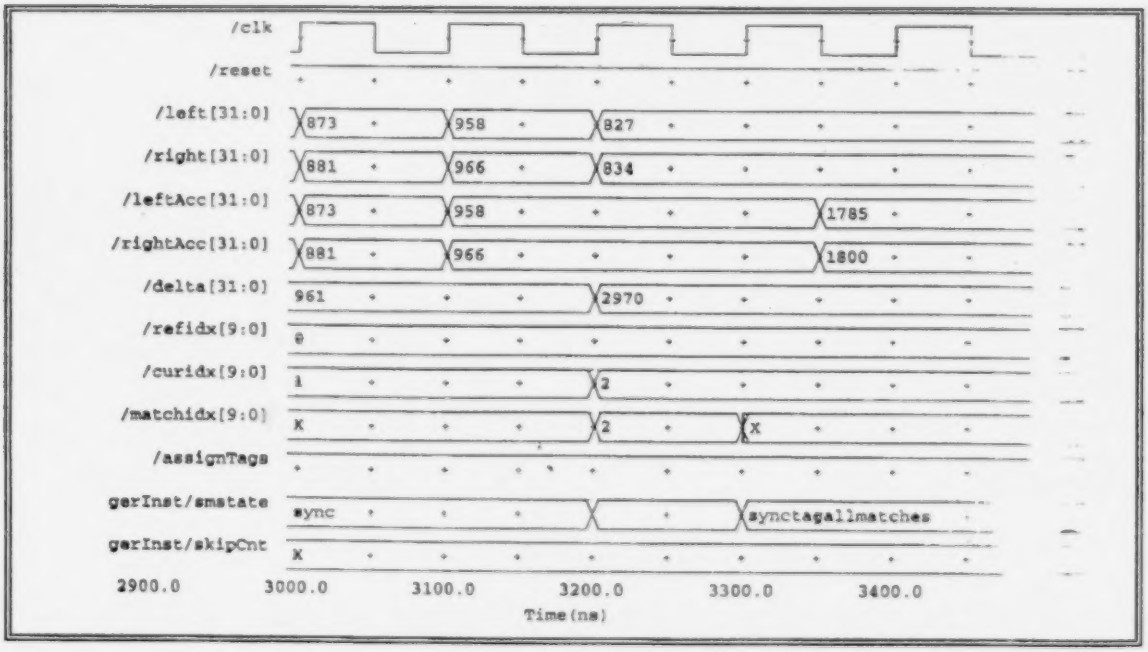


Figure 5-2 (ii) SIMULATION TIMING RESULTS

P148505.PDF [Page: 38 of 108]

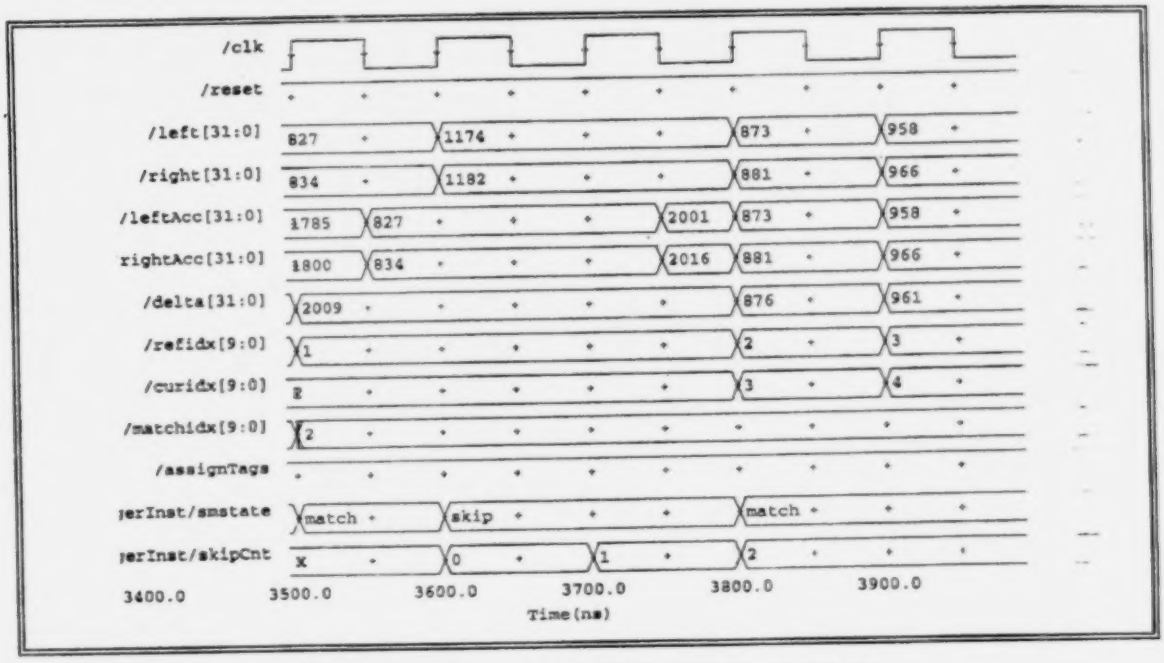


Figure 5-2 (iii) SIMULATION TIMING RESULTS

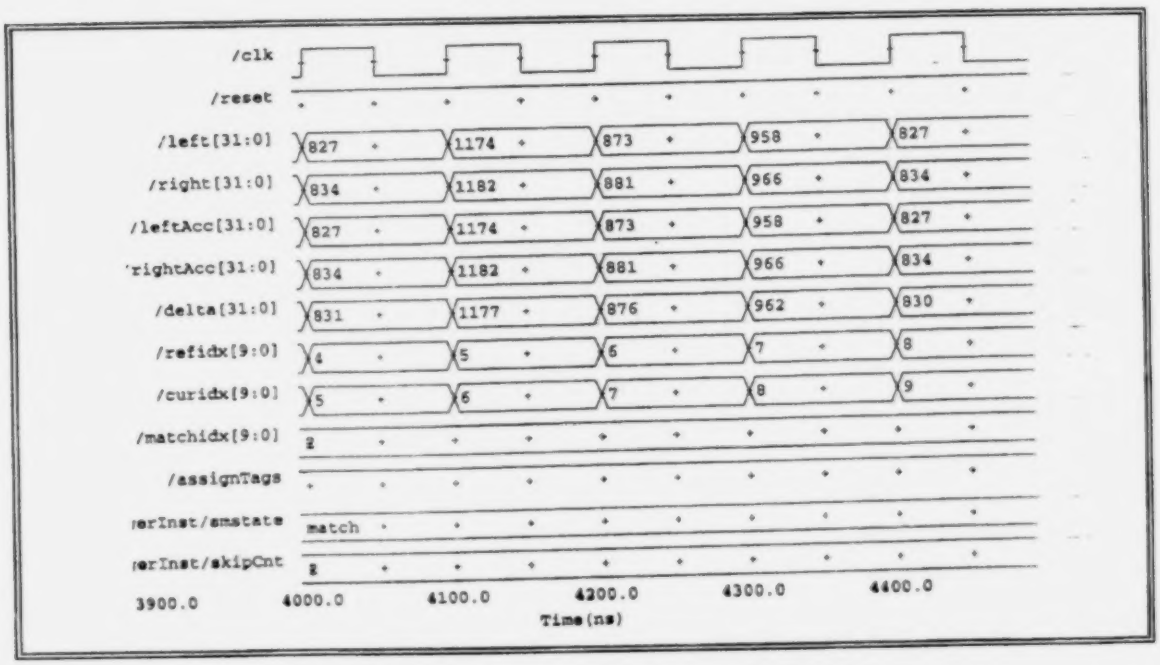


Figure 5-2 (iv) SIMULATION TIMING RESULTS

P148505.PDF [Page: 39 of 108]

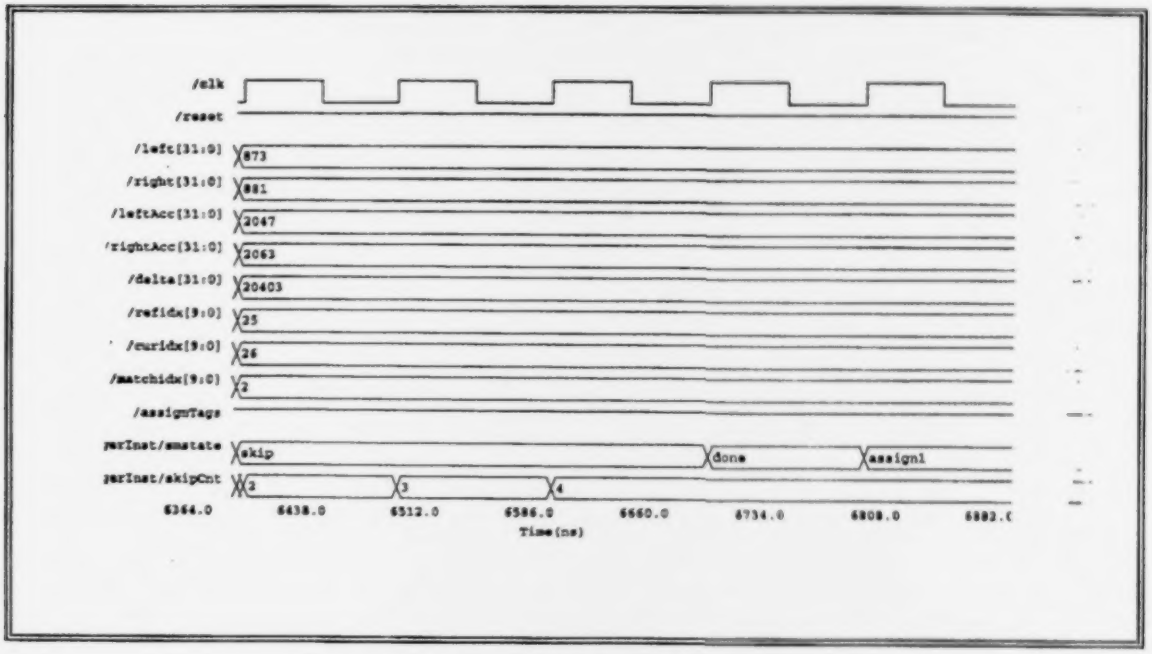


Figure 5-3 (i) SIMULATION TIMING RESULTS

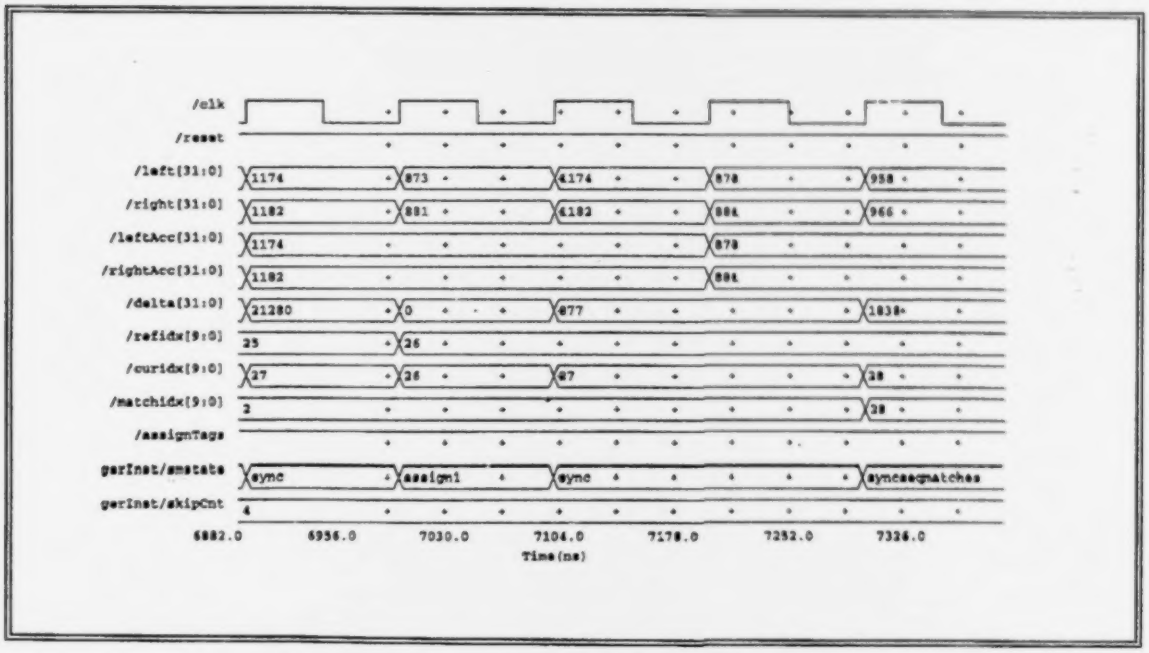


Figure 5-3 (ii) SIMULATION TIMING RESULTS

P148505.PDF [Page: 40 of 108]

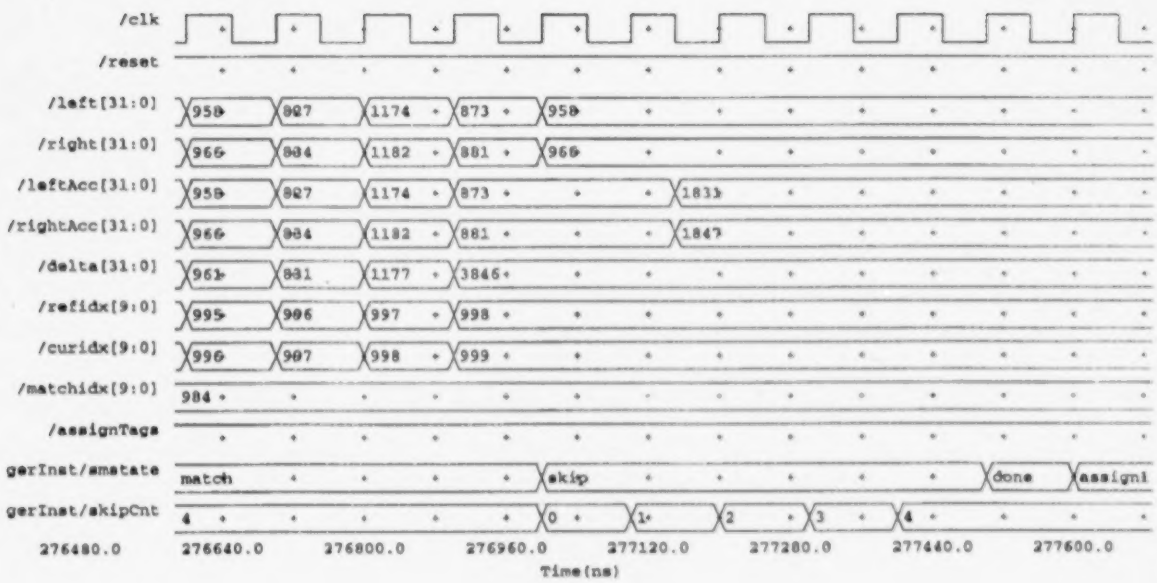


Figure 5-4 SIMULATION TIMING RESULTS

FUNCTION NUMBER	FUNCTION	STATE COUNT
1	addNextInterval	1
2	computeDelta	1
3	executeState	0
4	referenceCurrentPulse	1
5	referenceNextPulse	1
6	setNextInterval	1
7	switchToAcquisition	0
8	switchToLockon	0
9	deltaMatches	0
10	tagAt	1
11	deltaGreaterThanMaxT	0
12	deltaGreaterThanRight	0
13	deltaLessThanLeft	0
14	setupToSkip	0
15	missingPulseThresholdIsZero	0
16	matchCountGreaterThanMinMatchCount	0
17	accumulateTimeForMatchedPulsesLoop	2
18	incrementMatchCount	1
19	storeCurrentIndexInMatchList	1
20	skipCountLessThanMissingPulseThreshold	0
21	incrementSkipCount	1
22	accumulateTimeForNumberOfIntervalsLoop	1
23	setupPhase	1

Table 5-1 MONITORED SEQUENCE TAGGER -- STATE TRANSITION COUNTS
(Page 1 of 2)

P148505.PDF [Page: 42 of 108]

FUNCTION NUMBER	FUNCTION	STATE COUNT
24	initializeMatchCount	1
25	sequenceMatches	0
26	setStateToMatchLock	0
27	setStateToSync	0
28	match	0
29	matchLock	0
30	skipPulse	0
31	sync	0

Table 5-1 MONITORED SEQUENCE TAGGER -- STATE TRANSITION COUNTS
(Page 2 of 2)

P148505.PDF [Page: 43 of 108]

5.3 EXPERIMENTAL RESULTS (continued)

1000 pulses in the file. The histograms of Figure 5-5 indicate the total state transition counts for each of the fine grain functions described above. Counts for the addNextInterval function are shown on the left; counts for the sync function on the right. The order of histogram entries matches the functions listed in Table 5-1.

Of particular interest in these plots is the differences in the number of hits of the referenceCurrentPulse, setNextInterval and tagAt functions. In Figure 5-5(a), these functions are more dominant because the four level stagger sequence occurs in about 75% of the data, and in lock-on mode, the sequence tagger would execute these functions more often. Note that the match lock and sync functions are more dominant in the single interval case indicating that this tagger has remained in acquisition mode more often, trying to lock-on to the four level stagger (both taggers must consider all of the data in the pulse stream).

Histograms for Emitter E069 and E104 are shown in Figures 5-6 and 5-7 respectively. Even though these files contain data from more than one emitter, a tagger was selected for the predominant emitter in each case.

Histograms for Emitter 126 are shown in Figure 5-8. There are seven single interval pulse trains interleaved in this file. The relative height of the above mentioned functions indicate the relative dominance of each pulse train.

The total number of state transitions for each tagger in each file is shown in Table 5-2 below. These data is drawn from the results of each run listed in Appendix C.

From the data in Table 5-2, the average number of states per pulse per tagger is 5.5. Using a **50 ns clock** (state dwell time), the average execution time per pulse for a tagger is **275 ns**. This represents an **average pulse throughput of 3.6 million pulses per second**. Based on experience, about 90% of the coarse channels contain a single emitter and therefore only one tagger will be instantiated in order to deinterleave the pulse stream. Therefore, the pulse throughput of a sequence tagger implemented in hardware is estimated to be **in excess of one million pulses per second**.

P148505.PDF [Page: 44 of 108]

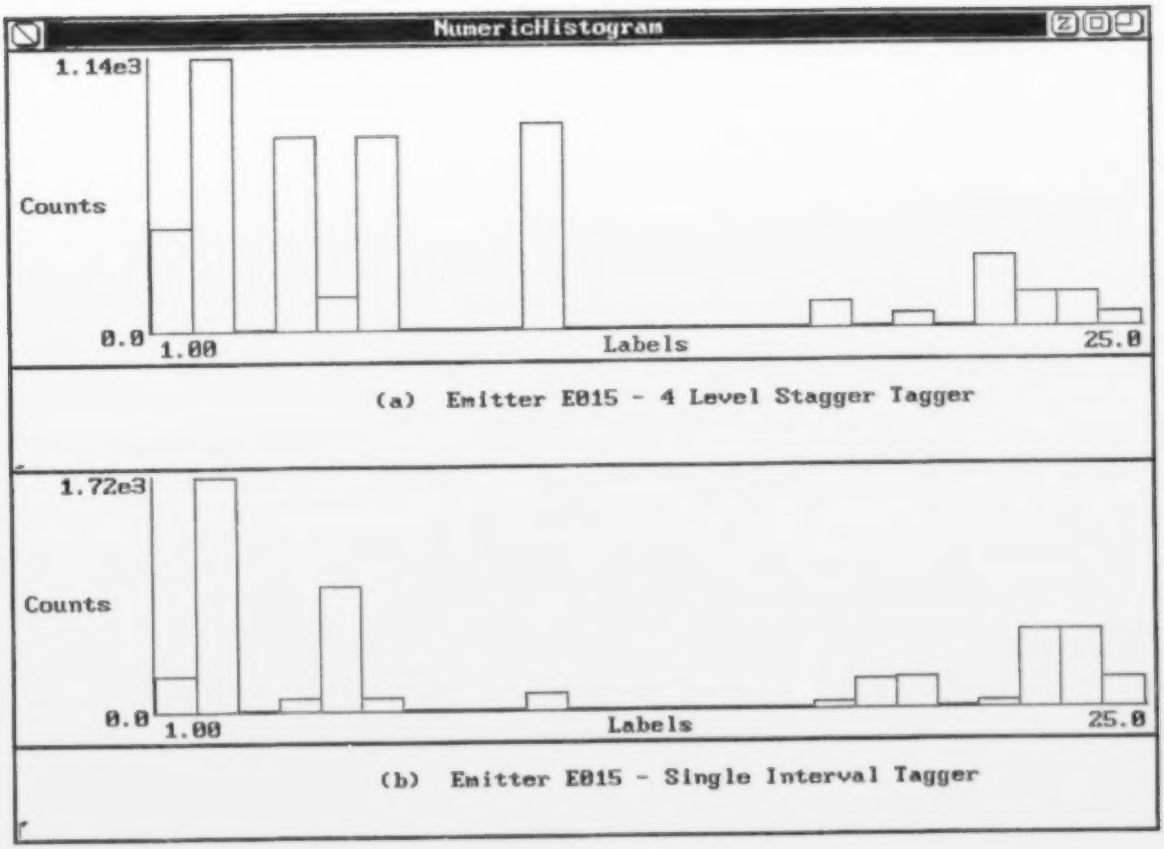


Figure 5-5 EMITTER E015

P148505.PDF [Page: 45 of 108]

Figure 5-6 EMITTER E069 -- SINGLE INTERVAL TAGGER

P148505.PDF [Page: 46 of 108]

Figure 5-7 EMITTER E104 -- SINGLE INTERVAL TAGGER

P148505.PDF [Page: 47 of 108]

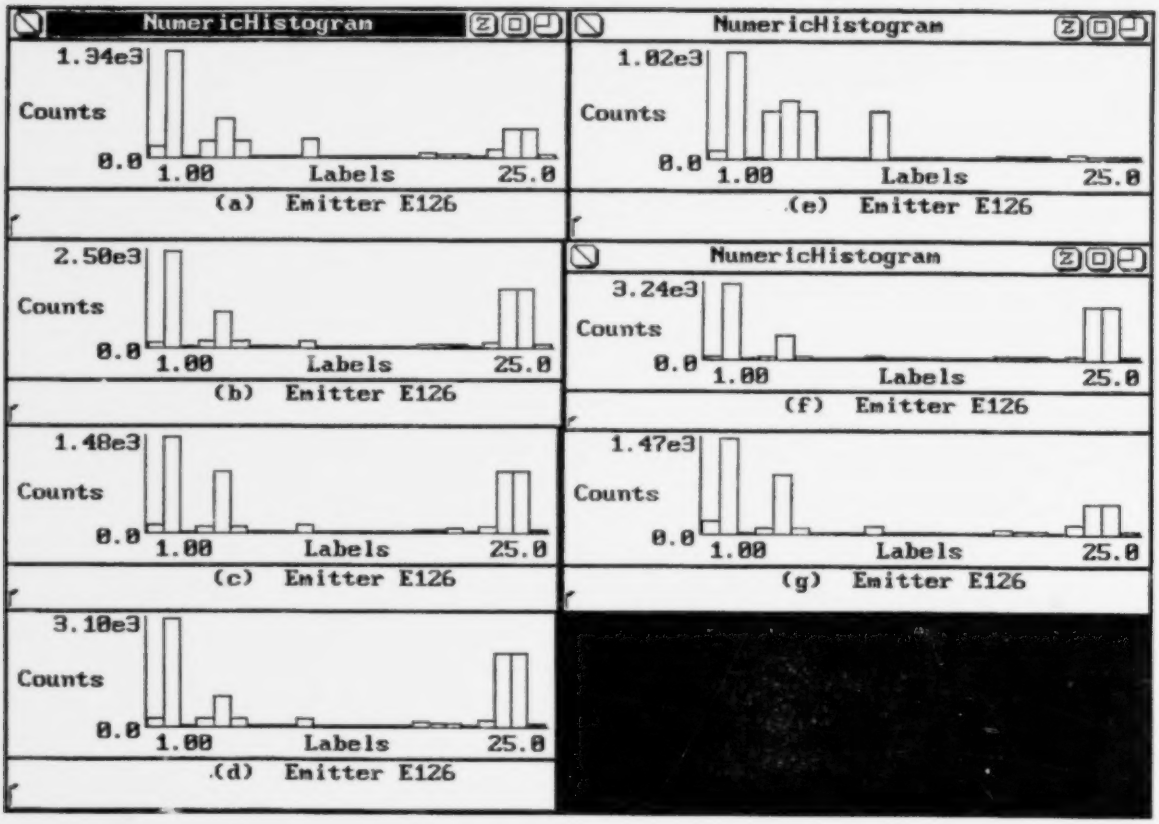


Figure 5-8 EMITTER E126

P148505.PDF [Page: 48 of 108]

TAGGER NO.	E015	E069	E104	E126
1	4921	5469	4851	3313
2	4914			6858
3				4725
4				9080
5				3045
6				8880
7				3863
Total States	9835	5469	4851	39764
Total Pulses	1000	1000	1000	1000
States per Pulse	9.8	5.5	4.9	39.8

Table 5-2 MEASURED STATE TRANSITIONS

P148505.PDF [Page: 49 of 108]

6.0 CONCLUSION

The Tagger Processing, part of the fine channelization algorithms performed in the AMEP preprocessor, can be realized in a single 6U x 160 mm hardware assembly using (primarily) programmable logic and standard high-speed memory devices. Two models of the algorithm (VHDL and Smalltalk) demonstrate that the real-time performance will meet the required pulse throughput rate. Event Monitoring algorithms are still in the development stage and although real-time performance will likely dictate that these algorithms must be implemented in custom hardware, significant technical risk would be associated with any realization in hardware at the current time. By providing a hardware prototype for tagging functions, the software load of fine channelization will be roughly halved which will permit faster development and testing of the AMEP software with longer data files.

6.1 ARCHITECTURE SELECTION

The Alternate Algorithm Architecture presented in Section 2.5 details a stream-oriented approach to supplying pulse data to the taggers. As was highlighted in the text, this technique may result in an a reduction of the number of pulses that go unclaimed by a tagger because the tagger can effectively remain locked-on from packet to packet (between main beam illuminations).

There is a cost associated with stream tagging in terms of real-estate. Tagging based on packet rules (packet processing) is a process that runs to completion; the stream tagging process is persistent. The impact of persistence is that any instance variables required in the tagging process must be stored while the tagger is suspended waiting on data. A second penalty associated with stream tagging results from the complexity of flow control and scheduling of the taggers in a channel. With block processing, packets are processed as they are formed. With stream-oriented tagging, all the taggers in a course channel must execute as each pulse arrives in the channel or an even more complex approach involving time slicing with context switching must be implemented.

The benefit of stream tagging outlined above was the reduction in the number of untagged pulses. The software development activities of the CANEWS 2 - Interim ADM Phase 1B development has identified the requirement to provide continuous monitoring of the presence of residue (untagged) pulses with either the block (packet) or stream tagging approaches. Figure 6-1 [ref 3] illustrates the desired approach to provide processing of residue pulses. In this design, pulses not claimed by the taggers in a course channel are extracted and monitored by the preprocessor where a special Residue Track is established to process these events. The residue processing is required to determine if unclaimed pulses belong to a new emitter, a new mode for the current emitter, or if they are simply a result of poor tagger performance caused by high occurrences of dropped pulses.

With residue pulse processing required regardless of how pulses are presented to the taggers, there is little, or no net gain from using the stream oriented approach. If the original packet processing technique is used, reduction in complexity of state machines, and a corresponding reduction in hardware real-estate can be realized.

P148505.PDF [Page: 50 of 108]

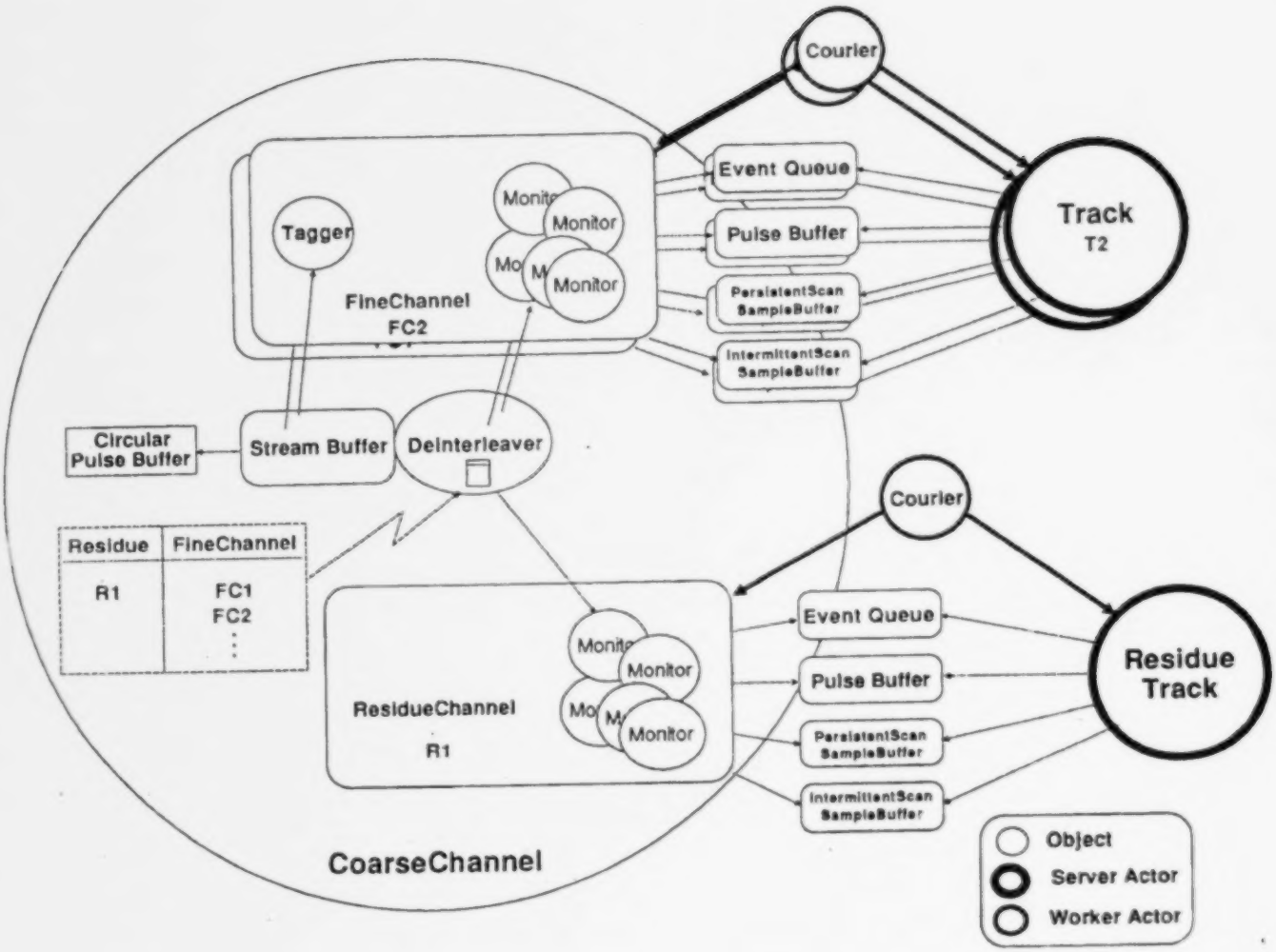


Figure 6-1 FINE CHANNELIZATION

P148505.PDF [Page: 51 of 108]

6.2 PROTOTYPE DEVELOPMENT

The recommended approach for implementation would be to provide an assembly which could be inserted into the existing data path of the AMEP preprocessor hardware. This could be placed between the Parameter Filter CCA and the Buffer Addressable Memory CCA. The existing interfaces to these assemblies would be respected to minimize cost. Event Monitoring could be performed in software using the data stored in the pulse buffers on the BAM. As the Event Monitoring algorithms mature, they could migrate to hardware in order to achieve the optimum throughput.

The block diagram for the fine channelizer hardware prototype is shown in Figure 6-2. The fine channelizer prototype must include the event buffer shown in Figure 3-1. These event buffers would be accessible over the VME Subsystem Bus (VSB) and the VSB interrupt would be [optionally] asserted with the presence of Event Messages (such as pulses in buffer).

6.2.1 Development Plan

The following paragraphs outline the work elements required to implement a practical prototype hardware assembly. Included is a description of the functionality that must be provided and the associated development activity. Each of the Logic Blocks detailed should be realized using standard commercial devices and programmable device technology (EPLD and/or FPGA) to permit minor algorithm changes in the future.

Pulse Stream Design

This involves the detailed design and implementation of the hardware required to provide the following functions:

- Asynchronous storage of pulse data from the Parameter Filter CCA.
- Formation of pulse packets using gap, duration, and maximum number of pulse rules.
- Delta TOA as 48-bit delta with 32-bit result in μsecond resolution.
- Pulse tagging.

Tagger State Machine Design

- Include RF intervals in tagging algorithm. If the RF of the received pulse falls within one of up to 16 (arbitrary) levels and RF tagging is enabled, then proceed with sequence tagging.
- Include multi-interval tagger algorithm. This algorithm is similar to sequence tagging without the requirement for the PRIs to occur in sequence order.

Deinterleaver Design

- Sort and transfer pulse data to fine channel bins (interface to the BAM).

P148505.PDF [Page: 52 of 108]

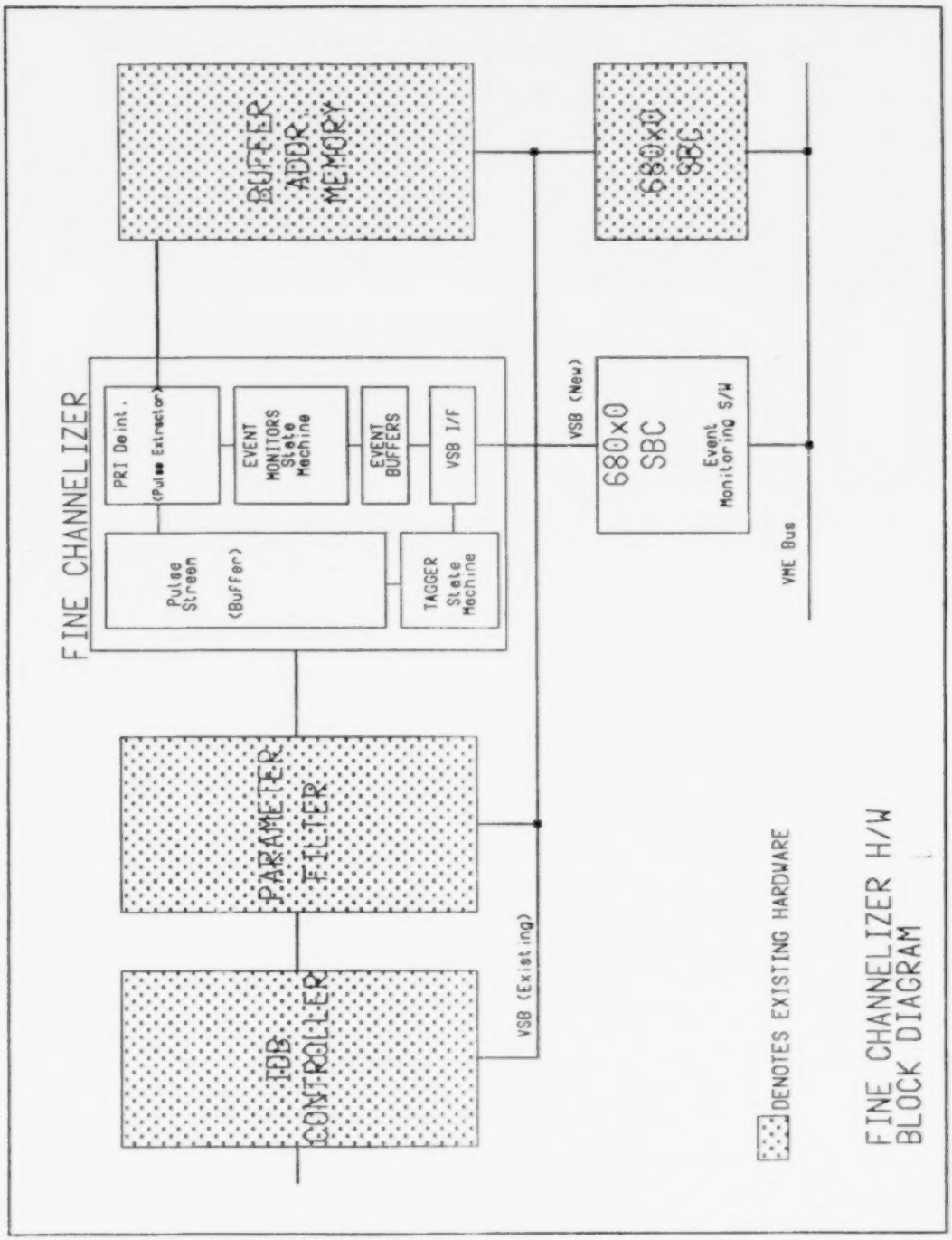


Figure 6-2 FINE CHANNELIZER H/W BLOCK DIAGRAM

P148505.PDF [Page: 53 of 108]

6.2.1 Development Plan (continued)

- Provide residue pulse analysis, sorting, and transfer to fine channel bins.

VSB Interface Design

- Support for A32/D32 slave operation.
- Support for VSB IRQ generation.
- Read of EVENT BUFFERS for each fine channel.
- Programming of tagger and deinterleaver parameters.

Circuit Card Assembly Design

This activity will include schematic capture, board layout (6U x 160 mm VME format), and manufacture preparation for the prototype assembly.

Circuit Card Assembly Build/Debug

This activity will include the build, bench test, and integration test with the AMEP target testbed using test software which exercises the fine channelizer hardware using pulse data files from the AMEP pulse database.

Fine Channelizer Test Procedure Software Development

This activity includes the design and development of the software required to exercise all of the fine channelizer hardware. This software will provide a testbed for manually defining taggers, injecting pulse data from a pulse data file into the preprocessor, reporting/recording the tagged pulse and residue data, and indicating a measure of the real-time performance of the algorithm.

Final Report

A final report detailing the theory of operation and specifying the measured performance of the prototype hardware shall be generated for the final design review.

A complete drawing package for the assembly and all VHDL source code would also be delivered at interim and final design reviews.

P148505.PDF [Page: 54 of 108]

7.0 REFERENCES

[1] "Software Requirements Specification for IAMEP CSCI of the CANEWS 2 Interim ADM", Lockheed Canada #1000-45-016 Version 01, 22 March 1992.

[2] "CANEWS 2 Software Algorithm Manual", SKL Document #1000-45-049.01.0, 27 March 1992, SECRET.

[3] "A New Top Level Architecture for the Interim AMEP CSCI", Software Kinetics Document # 1000-600001-012, 18 March 1994.

P148505.PDF [Page: 55 of 108]

Appendix A

StreamBuffer and SequenceStreamTagger

(Smalltalk Class Fileout)

```
Array variableSubclass: #StreamBuffer
  instanceVariableNames: 'readIndex writeIndex size '
  classVariableNames: ''
  poolDictionaries: ''
!

StreamBuffer comment: '
A streamBuffer is a FIFO which is associated with a particaular coarse
channel.  The streamBuffer is analyzed by each tagger within the channel.
A reference count is maintained on each pulse and is manipulated by the
taggers.  In order to extract a pulse from the streamBuffer, the reference
count must be 0.

A streamBuffer is created for a fixed size using the new: aSize method.
Pulses are added to the stream using the add: method.  Pulses are removed
using the remove method.  Each tagger uses the remaining public functionality.

Instance Variables

    readIndex:  an index pointing to the first pulse in the FIFO.
    writeIndex: an index pointing to one after the last entry of the FIFO.
    size:       the number of objects in the FIFO.'
!

!StreamBuffer class publicMethods !

new: size
"
    Answer a new streamBuffer of size: size.
"
    ^ (super new: size) initialize; yourself.
!

test
"
    StreamBuffer test.
"
    | sb |

    sb := self new: 100.
    sb add: 1.
    sb add: 2.

    Transcript cr; show: (Time millisecondsToRun: [
        100000 timesRepeat: [ sb nextIndexFrom: 1 ] ]) printString.

    ^ sb.
! !

!StreamBuffer publicMethods !

add: anObject
"
    If the streamBuffer is not full, append anObject.  The streamBuffer
    implements a FIFO.
"
  size < super size
        ifTrue: [
            self at: writeIndex put: anObject.
            size := size + 1.
            (writeIndex := writeIndex + 1) > super size
                ifTrue: [ writeIndex := 1 ] ].
```

```
!

contains: anIndex
"
    Answer true of anIndex points to a valid object within the streamBuffer.
"

    writeIndex >= readIndex
        ifTrue: [
            (anIndex >= readIndex and: [ anIndex < writeIndex ])
                ifTrue: [ ^true ] ]
        ifFalse: [
            (anIndex >= readIndex and: [ anIndex <= super size ])
                ifTrue: [ ^true ].
            anIndex < writeIndex
                ifTrue: [ ^true ] ].
    ^false.
!

firstIndex
"
    Answer the index of the first object in the streamBuffer.
"
    ^readIndex.
!

nextIndexFrom: aReadIndex
"
    Answer the next index with respect to aReadIndex in the streamBuffer.
"
    (aReadIndex + 1) > super size
        ifTrue: [ ^1 ].
    ^aReadIndex + 1.
!

previousIndexFrom: aReadIndex
"
    Answer the previous index with respect to aReadIndex in the streamBuffer.
"
    (aReadIndex - 1) < 1
        ifTrue: [ ^super size ].
    ^aReadIndex - 1.
!

remove
"
    If the streamBuffer is not empty and the object is not referenced,
    extract and answer the object.  Else, answer nil.

    Note: the object must implement a reference count mechanism and therefor
    respond to notReferenced.
"
    | aPulse |

    (size > 0 and: [ (aPulse := self at: readIndex) notReferenced ])
        ifTrue: [
            (readIndex := readIndex + 1) > super size
                ifTrue: [ readIndex := 1 ].
            size := size - 1.
            ^ aPulse ].
    ^nil.
!

size
```

```
"
    Answer the number of objects within the streamBuffer.
"
    ^size.
!

tagWith: anId at: anIndex
"
    Add a tag to the object at anIndex to the object in the stream.  The
    id of the tag is anId.

    Note: The objects in the stream must have a tagging mechanism and
    respond to addTag:.
"
    (self at: anIndex) addTag: anId.
! !

!StreamBuffer privateMethods !

initialize
"
    Initialize the streamBuffer.
"
    readIndex  := 1.
    writeIndex := 1.
    size := 0.
! !


Object subclass: #SequenceStreamTagger
  instanceVariableNames: 'id groupIntervalLeft groupIntervalRight priLeft priRig
ht matchedPulses maxT phase missingPulseThreshold minMatchCount numberOfInterval
s streamBuffer currentIndex streamReference matchCount left right skipCount delt
a state '
  classVariableNames: ''
  poolDictionaries: ''
!

SequenceStreamTagger comment: '
    The sequenceTagger assigns tags to a pulse train within a trackChannel
    using a knowledge of the pri sequence as well as the stagger intervals
    and a set of known rf intervals.

    The tagger supports repeated intervals and can be set up to predict over
    an adjustable number of skipped pulses. An acquisition mode is provided
    with a threshold indicating the minimum number of matches that
    are required before lock-on can be acheived. If acquisition is abandoned
    because the number of sequential matches have not been obtained, the
    oldest pulse that is referenced in the stream for this acquisition search
    is left untagged and the next oldest is considered as a reference for
    a repeated acquisition.

    Once lockon has been acheived, the pulse tagging proceeds until the number
    of missing pulses detected exceeds a threshold at which point the
    acquisition mode is re-entered.

    A sequence tagger is created and initialized with its tagging parameters
    using the following methods:

        openOn: aTaggerDescriptor creates a new instance of a
                SequenceStreamTagger on: aStreamBuffer.

    The algorithm is executed using the assignTags method.
```

```
    Class Variables: (None)

    Instance Variables:

    id: the id number of this tagger.
    groupIntervalLeft: the minimum group interval time.
    groupIntervalRight: the maximum group interval time.
    priLeft: the left pri intervals in the sequence.
    priRight: the right pri intervals in the sequence.
    matchedPulses: an array of pulse indicies that matched during acquisition.
    maxT: right extreme of the largest interval in the sequence.
    phase: the current phase index to the interval sequence.
    missingPulseThreshold: the number of allowable missing pulses in lockon.
    minMatchCount: the minimum number of sequentially matched pulse to switch
                   to lockon mode.
    numberOfIntervals: the number of intervals in the sequence.
    streamBuffer: a pointer to the streamBuffer for the coarse channel.
    currentIndex: a pointer to the the newest pulse in the stream which
                  is being considered by this tagger.
    streamReference: a pointer to the oldest pulse in the stream which
                     is being considered by this tagger.
    matchCount: the current number of correct matches during acquisition.
    left: the current left prediction value.
    right: the current right prediction value.
    skipCount: the number of skipped intervals.
    delta: the toa difference between the current and referenced pulse.
    state: the operating state of the algorithm.


    Global Variables: (None)

    Pool Dictionaries: (None)'
!

!SequenceStreamTagger class publicMethods !

openOn: aTaggerDescriptor for: aStreamBuffer
"
    Answer a new sequenceStreamTagger tuned on aTaggerDescriptor to
    operate on aStreamBuffer.
"
    ^ self new initializeOn: aTaggerDescriptor for: aStreamBuffer;
              yourself.
! !

!SequenceStreamTagger class privateMethods !

defaultMaxMissingPulses
"
    Answer the default maximum number of allowable missing pulses.
"
    ^ 20.
!

defaultMinMatchCount
"
    Answer the default minimum match count in order to achieve lockon.
"
    ^ 2.
! !

!SequenceStreamTagger publicMethods !

assignTags
```

```
"
    Process control method.

    If called the first time, initialize the streamReference and
    currentIndex to the oldest pulse in the streamBuffer.

    Iterate over the streamBuffer.  Determine the delta TOA between the
    streamReference pulse and the current pulse.  Execute the part of the
    tagging algorithm defined by state. Once the existing pulses in the
    streamBuffer have been processed, exit so that more pulses can be added
    to the stream.

        SequenceStreamTagger testOn: PulseStream open for: 1000.
        SequenceTagger testOn: PulseStream open for: 1000.
        SequenceStreamTagger testPulseByPulseOn: PulseStream open for: 1000.
"

    streamReference isNil
        ifTrue: [
            streamReference := streamBuffer firstIndex.
            (streamBuffer at: streamReference) reference.
            currentIndex := streamBuffer nextIndexFrom: streamReference ].

    [ streamBuffer contains: currentIndex ] whileTrue: [

        delta := (streamBuffer at: currentIndex)
                    toaDifferenceAsus: (streamBuffer at: streamReference).

        self executeState.
        currentIndex := streamBuffer nextIndexFrom: currentIndex ].
!

id
"
    Answer the tag id of this tagger.
"
    ^id
!

streamBuffer: aStreamBuffer
"
    Set the streamBuffer to aStreamBuffer.
"
    streamBuffer := aStreamBuffer.
!

updateWith: aTaggerDescriptor
"
    Update a tagger with aTaggerDescriptor as the new set of parameters.
    For any nil parameter, leave the corresponding paramater in the tagger
    unchanged.

    A change of any interval in either the rf or priIntervals collection
    requires that the entire collection be present in the descriptor.
"
    | priIntervals minT sumOfDeltas |

    aTaggerDescriptor priIntervals isEmpty
        ifFalse: [
            priIntervals := aTaggerDescriptor priIntervals.
            numberOfIntervals := priIntervals size.

            priLeft := Array new: numberOfIntervals.
            priRight := Array new: numberOfIntervals.
```

```
            maxT := 0.
            groupIntervalLeft := 0.
            groupIntervalRight := 0.

            sumOfDeltas := 0.
            1 to: numberOfIntervals do: [ :index |
                priLeft at: index put: (priIntervals at: index) left.
                priRight at: index put: (priIntervals at: index) right.
                maxT := maxT max: (priIntervals at: index) right.
                groupIntervalLeft :=
                    groupIntervalLeft + (priIntervals at: index) left.
                groupIntervalRight :=
                    groupIntervalRight + (priIntervals at: index) right.
                minT = nil
                    ifTrue: [ minT := (priIntervals at: index) left ].
                minT := minT min: (priIntervals at: index) left.
                sumOfDeltas := sumOfDeltas + (priIntervals at: index) width ]].

    aTaggerDescriptor missingPulseThreshold isNil
        ifFalse: [ missingPulseThreshold :=
                        aTaggerDescriptor missingPulseThreshold ].
    aTaggerDescriptor minMatchCount isNil
        ifFalse: [
            minMatchCount := aTaggerDescriptor minMatchCount.
            matchedPulses := Array new: minMatchCount ].
! !

!SequenceStreamTagger privateMethods !

addNextInterval
"
    Increment the phase of the interval set.  Add the next left and right
    intervals to the existing left and right interval values.
"

    (phase := phase + 1) > numberOfIntervals ifTrue: [ phase := 1 ].
    left  := left  + (priLeft  at: phase).
    right := right + (priRight at: phase).
!

executeState
"
    Execute a method corresponding to the symbol in state.  The states can be
    #sync, #matchLock, #lockon or #skipPulse.
"
    state = #sync
        ifTrue: [
            self sync.
            ^self ].
    state = #matchLock
        ifTrue:  [
            self matchLock.
            ^self ].
   state = #lockon
        ifTrue: [
            self match.
            ^self ].
    state = #skipPulse
        ifTrue: [
            self skipPulse.
            ^self ].
!

initializeOn: aTaggerDescriptor for: aStreamBuffer
```

```
"
    Initialize the tagger on aTaggerDescriptor for a streamBuffer.  The
    streamBuffer is unique for each coarse channel.
"
    | priIntervals minT sumOfDeltas |

    priIntervals := aTaggerDescriptor priIntervals.
    numberOfIntervals := priIntervals size.

    id := aTaggerDescriptor tagId.
    priLeft := Array new: numberOfIntervals.
    priRight := Array new: numberOfIntervals.
    maxT := 0.
    groupIntervalLeft := 0.
    groupIntervalRight := 0.

    sumOfDeltas := 0.
    1 to: numberOfIntervals do: [ :index |
        priLeft at: index put: (priIntervals at: index) left.
        priRight at: index put: (priIntervals at: index) right.
        maxT := maxT max: (priIntervals at: index) right.
        groupIntervalLeft :=
            groupIntervalLeft + (priIntervals at: index) left.
        groupIntervalRight :=
            groupIntervalRight + (priIntervals at: index) right.
        minT = nil
            ifTrue: [ minT := (priIntervals at: index) left ].
        minT := minT min: (priIntervals at: index) left.
        sumOfDeltas := sumOfDeltas + (priIntervals at: index) width ].

    aTaggerDescriptor missingPulseThreshold isNil
        ifTrue: [
           sumOfDeltas = 0
                ifTrue: [
                    missingPulseThreshold :=
                            self class defaultMaxMissingPulses ]
                ifFalse: [
                    missingPulseThreshold :=
                        (self class defaultMaxMissingPulses min: (
                            (minT // sumOfDeltas) * numberOfIntervals))
                                max: 2 ] ]
        ifFalse: [
            missingPulseThreshold :=
                aTaggerDescriptor missingPulseThreshold ].

    aTaggerDescriptor minMatchCount isNil
        ifTrue: [
            minMatchCount := numberOfIntervals -
                aTaggerDescriptor nbrUniquePriIntervals +
                                        self class defaultMinMatchCount ]
        ifFalse: [
            minMatchCount := aTaggerDescriptor minMatchCount ].

    matchedPulses := Array new: minMatchCount.

    streamBuffer := aStreamBuffer.
    state := #sync.
!

match
"
    If the state is #match, this method is called for each iteration of the
    streamBuffer.  Begin by comparing the delta TOA to the current predicted
    interval bounded by left and right.  If the compare is successful, tag
```

```
    the pulse associated with the currentIndex, set the streamReference
    to the currentIndex and find the next interval in the sequence.

    If the compare is not sucessful and if delta is less than the left
    predicted interval, exit allowing the next pulse in the stream to be
    processed.  This condition occurs when a pulse from another emitter is
    encountered in a TOA position which is between the pulses which this
    tagger is tagging.

    If this second test fails, there is at least one missing pulse.  If the
    missingPulseThreshold is zero (no missing pulses are allowed), switch
    states to acquisition and exit (lockon is lost).

    If one or more missing pulses are allowed, add the next predicted interval
    to the current interval, set up for missing pulse processing by setting
    the state to #skipPulses, move the currentIndex back one (it will be
    advanced one by the control method) and exit.  The control method will
    then force the execution of the skipPulse algorithm as many times as
    required to match the current pulse or lose lockon.
"

    ( delta >= left and: [ delta <= right ] )
        ifTrue:  [
            streamBuffer tagWith: id at: currentIndex.
            self referenceCurrentPulse.
            self setNextInterval.
            ^self ].
    delta < left
        ifTrue: [ ^self ].

    missingPulseThreshold = 0
        ifTrue: [
            self switchToAcquisition.
            ^self ].

    self addNextInterval.
    state := #skipPulse.
    skipCount := 0.
    currentIndex := streamBuffer previousIndexFrom: currentIndex.
!

matchLock
"
    Determine if delta lies within the predicted interval.  If true, determine
    if the total number of matches so far meets the minimumMatchCount
    threshold.  If this is the case, switch to the lockon mode and exit.

    If either of the above tests fail and if the delta is greater than the
    predicted right interval, the lockon attempt has failed.  Move the stream
    reference to the next pulse and set the state to #sync to restart the
    search for lockon.

    If all of the above tests fail, remain in the #matchLock state so that
    the next interval in the pulse stream can be tested.
"

    ( delta >= left and: [ delta <= right ] )
        ifTrue: [
            self sequenceMatches
                ifTrue: [
                    self setNextInterval.
                    self switchToLockon.
                    ^ self ].
            self addNextInterval.
```

```
            ^self ].
    delta > right
        ifTrue: [
            self referenceNextPulse.
            state := #sync ].
!

referenceCurrentPulse
"
    Dereference the current streamReference pulse, set the streamReference
    to the currentIndex and establish a reference on that pulse.  A reference
    count is maintained by the streamBuffer to establish referencing.
"
    (streamBuffer at: streamReference) deReference.
    streamReference := currentIndex.
    (streamBuffer at: streamReference) reference.
!

referenceNextPulse
"
    Dereference the current streamReference pulse, set the streamReference
    to the next pulse and establish a reference on that pulse.  A reference
    count is maintained by the streamBuffer to establish referencing.  Set
    the currentIndex to the streamReference.
"
    (streamBuffer at: streamReference) deReference.
    streamReference := streamBuffer nextIndexFrom: streamReference.
    (streamBuffer at: streamReference) reference.
    currentIndex := streamReference.
!

sequenceMatches
"
    Answer true if the stream fragment matches the currect number of intervals
    in the known PRI sequence.  This method is entered if a TOA interval
    matches the current interval.

    Store the index of the current pulse in an array.  If the matchCount
    meets the minMatchCount, tag the streamReferenced pulse, tag all other
    pulses which matched the sequence, and return true.

    If the minMatchCount has not been exceeded, increment the matchCount
    and return false.
"
    matchedPulses at: matchCount put: currentIndex.
    matchCount >= minMatchCount
        ifTrue: [
            streamBuffer tagWith: id at: streamReference.
            1 to: matchCount do: [ :pulseIndex |
                streamBuffer tagWith: id at: (matchedPulses at: pulseIndex) ].
            ^true ].
    matchCount := matchCount + 1.
    ^false.
!

setNextInterval
"
    Increment the phase of the interval set.  Set the existing right
    and left intervals to the next predicted interval.
"

    (phase := phase + 1) > numberOfIntervals ifTrue: [ phase := 1 ].
    left  := priLeft  at: phase.
    right := priRight at: phase.
```

```
!

skipPulse
"
    Execute if a missing pulse has been detected during lockon. While the
    skipCount does not exceed the missingPulseThreshold, test the current
    TOA interval against each additional predicted interval in the sequence.
    If a match can be found, tag the current pulse, set the predicted interval
    to the next interval in the sequence, switch the state back to #lockon
    and exit.

    If the current TOA interval is less than a current predicted interval,
    exit so that the next pulse can be obtained.  This occurs when a pulse
    from another emitter is interleaved.

    If the missingPulseThreshold is exceeded, switch to acquisition mode.
"
    [ skipCount < missingPulseThreshold ]
        whileTrue: [

            ( delta >= left and: [ delta <= right ] )
                ifTrue: [
                    streamBuffer tagWith: id at: currentIndex.
                    self setNextInterval.
                    self switchToLockon.
                    ^self ].
            delta < left
                ifTrue: [ ^self ].

            self addNextInterval.
            skipCount := skipCount + 1 ].

    self switchToAcquisition.
!

switchToAcquisition
"
    Set the streamReference to the next pulse in the stream.  Set the state
    to @sync.
"
    self referenceNextPulse.
    state := #sync.
!

switchToLockon
"
    Set the streamReference to the currentIndex and switch the state to
    #lockon.
"
    self referenceCurrentPulse.
    state := #lockon.
!

sync
"
    Sync is the initial phase of acquisition.  If the curent delta is greater
    than maxT ( the largest right value of all of the intervals in the known
    sequence ), set the streamReference to the next pulse ( the first interval
    is not one in the seqence ).

    If the current interval matches one of the intervals in the known sequence,
    set the phase to the index of that interval, set the matchCount to one
    and determine if the sequence matches (if the matchCount meets the
    minMatchCount requirement).
```

```
    If the sequence matches, determine the next prediction interval from
    the sequence and switch to lockon.

    If the sequence does not match, get the next prediction window and
    switch to the matchLock state where subsequent pulses will be matched.

    If none of the intervals match, exit so that the interval associated with
    the next pulse can be tested.

    delta > maxT
        ifTrue: [
            self referenceNextPulse.
            ^self ].

    1 to: numberOfIntervals do: [ :priIndex |
        ( delta >= (priLeft at: priIndex)
                                and: [ delta <= (priRight at: priIndex) ] )
            ifTrue: [
                phase := priIndex.
                matchCount := 1.
                self sequenceMatches
                    ifTrue: [
                        self setNextInterval.
                        self switchToLockon.
                        ^self ].

                left  := priLeft  at: phase.
                right := priRight at: phase.
                self addNextInterval.

                state := #matchLock.
                ^ self ] ].
! !
```

P148505.PDF [Page: 67 of 108]

Appendix B

MonitoredStreamBuffer

(Smalltalk Class Fileout)

```
Object subclass: #MonitoredSequenceStreamTagger
  instanceVariableNames: 'monitor id groupIntervalLeft groupIntervalRight priLef
t priRight matchedPulses maxT phase missingPulseThreshold minMatchCount numberOf
Intervals streamBuffer currentIndex streamReference matchCount left right skipCo
unt delta state '
  classVariableNames: 'MonitorConstants '
  poolDictionaries: ''
!

MonitoredSequenceStreamTagger comment: '
    The sequenceTagger assigns tags to a pulse train within a trackChannel
    using a knowledge of the pri sequence as well as the stagger intervals
    and a set of known rf intervals.

    The tagger supports repeated intervals and can be set up to predict over
    an adjustable number of skipped pulses. An acquisition mode is provided
    with a threshold indicating the minimum number of matches that
    are required before lock-on can be acheived. If acquisition is abandoned
    because the number of sequential matches have not been obtained, the
    oldest pulse that is referenced in the stream for this acquisition search
    is left untagged and the next oldest is considered as a reference for
    a repeated acquisition.

    Once lockon has been acheived, the pulse tagging proceeds until the number
    of missing pulses detected exceeds a threshold at which point the
    acquisition mode is re-entered.

    A sequence tagger is created and initialized with its tagging parameters
    using the following methods:

        openOn: aTaggerDescriptor creates a new instance of a
                MonitoredSequenceStreamTagger on: aStreamBuffer.

    The algorithm is executed using the assignTags method.

    Class Variables: (None)
```

```
    Instance Variables:


    id: the id number of this tagger.

    groupIntervalLeft: the minimum group interval time.

    groupIntervalRight: the maximum group interval time.

    priLeft: the left pri intervals in the sequence.

    priRight: the right pri intervals in the sequence.

    matchedPulses: an array of pulse indicies that matched during acquisition.

    maxT: right extreme of the largest interval in the sequence.

    phase: the current phase index to the interval sequence.

    missingPulseThreshold: the number of allowable missing pulses in lockon.

    minMatchCount: the minimum number of sequentially matched pulse to switch

                   to lockon mode.

    numberOfIntervals: the number of intervals in the sequence.

    streamBuffer: a pointer to the streamBuffer for the coarse channel.
    currentIndex: a pointer to the the newest pulse in the stream which
                  is being considered by this tagger.
    streamReference: a pointer to the oldest pulse in the stream which
                     is being considered by this tagger.
    matchCount: the current number of correct matches during acquisition.
    left: the current left prediction value.
    right: the current right prediction value.
    skipCount: the number of skipped intervals.
    delta: the toa difference between the current and referenced pulse.
    state: the operating state of the algorithm.


    Global Variables: (None)


    Pool Dictionaries: (None)'
!

!MonitoredSequenceStreamTagger class publicMethods !

initialize
"
    MonitoredSequenceStreamTagger initialize.

"
    MonitorConstants := Dictionary new.

    MonitorConstants
        at: #addNextInterval                    put:  1 @ 1;
        at: #computeDelta                       put:  2 @ 1;
        at: #executeState                       put:  3 @ 0;
```

```
        at: #referenceCurrentPulse                    put:  4 @ 1;
        at: #referenceNextPulse                       put:  5 @ 1;
        at: #setNextInterval                          put:  6 @ 1;
        at: #switchToAcquisition                      put:  7 @ 0;
        at: #switchToLockon                           put:  8 @ 0;

        at: #deltaMatches                             put:  9 @ 0;
        at: #tagAt                                    put: 10 @ 1;
        at: #deltaGreaterThanMaxT                     put: 11 @ 0;
        at: #deltaGreaterThanRight                    put: 12 @ 0;
        at: #deltaLessThanLeft                        put: 13 @ 0;
        at: #setupToSkip                              put: 14 @ 0;
        at: #missingPulseThresholdIsZero              put: 15 @ 0;
        at: #matchCountGreaterThanMinMatchCount       put: 16 @ 0;
        at: #accumulateTimeForMatchedPulsesLoop       put: 17 @ 2;
        at: #incrementMatchCount                      put: 18 @ 1;
        at: #storeCurrentIndexInMatchList             put: 19 @ 1;
        at: #skipCountLessThanMissingPulseThreshold   put: 20 @ 0;
        at: #incrementSkipCount                       put: 21 @ 1;
        at: #accumulateTimeForNumberOfIntervalsLoop   put: 22 @ 1;
        at: #setupPhase                               put: 23 @ 1;
        at: #initializeMatchCount                     put: 24 @ 1;
        at: #sequenceMatches                          put: 25 @ 0;
        at: #setStateToMatchLock                      put: 26 @ 0;
        at: #setStateToSync                           put: 27 @ 0;

        at: #match                                    put: 28 @ 0;
        at: #matchLock                                put: 29 @ 0;
        at: #skipPulse                                put: 30 @ 0;
        at: #sync                                     put: 31 @ 0.

    ^MonitorConstants
!

initializeForCounting
"
    MonitoredSequenceStreamTagger initialize.
    MonitoredSequenceStreamTagger initializeForCounting.

"
    MonitorConstants keysDo: [ :aKey |
        MonitorConstants at: aKey put: (((MonitorConstants at: aKey) x) @ 1) ].
    ^MonitorConstants
!

monitorConstants
"
"
    ^MonitorConstants.
!

openOn: aTaggerDescriptor for: aStreamBuffer
"
    Answer a new sequenceStreamTagger tuned on aTaggerDescriptor to
    operate on aStreamBuffer.
"
    ^ self new initializeOn: aTaggerDescriptor for: aStreamBuffer;
              yourself.
! !

!MonitoredSequenceStreamTagger class privateMethods !

defaultMaxMissingPulses
"
```

```
    Answer the default maximum number of allowable missing pulses.
    ^ 20.
!

defaultMinMatchCount
    Answer the default minimum match count in order to achieve lockon.
    ^ 2.
!

printToasOn: simPulse for: nPulses
    Notes:
        MonitoredSequenceStreamTagger printToasOn: PulseStream open for: 200.
    | simPulse p index streamBuffer fname oldToa |
    streamBuffer := StreamBuffer new: self streamBufferSize.
    fname := simPulse simFileStream file name.
    ( p := simPulse pulse ) isNil
        ifTrue: [ ^self ].
    index := 1.
    [ streamBuffer add: (TaggedStreamPulse newOn: p).
      index := index + 1.
      p := simPulse pulse ) notNil and: [ index <= nPulses ] ) ]
        whileTrue: [ ].
    index := index - 1.
    Transcript cr; cr; show: 'TOA Data for: ', fname.
    Transcript cr; cr; show: 'Pulse      TOA      Difference'.
    index := 1.
    streamBuffer do: [ :aPulse |
        oldToa isNil
            ifTrue: [ oldToa := aPulse toaAsus ].
        Transcript cr; show: (index printPaddedTo: 4), '   '.
        Transcript show: (aPulse toaAsus printPaddedTo: 9), '   ',
                         ((aPulse toaAsus - oldToa) printPaddedTo: 9).
        oldToa := aPulse toaAsus.
        index := index + 1 ].

    streamBufferSize
        "the size of the StreamBuffer.
        ^ 32

    testOn: nPulses for: nPulses
        "
        MonitoredSequenceStreamTagger testOn: PulseStream open for: 1000.
    | simPulse p index streamBuffer taggers nPulses fname
      firstTagger tds histograms count |
    streamBuffer := StreamBuffer new: self streamBufferSize.
    fname := simPulse simFileStream file name.
```

```
taggers := OrderedCollection new.

( p := simPulse pulse ) isNil
    ifTrue: [ ^self ].

index := 1.
[   streamBuffer add: (TaggedStreamPulse newOn: p).
    index := index + 1.
    (( p := simPulse pulse ) notNil and: [ index <= nPulses ] ) ]
        whileTrue: [ ].
index := index - 1.

(tds := SequenceStreamTaggerTest makeTaggersOnFile: fname) do: [ :aTd |
    taggers addAll: (aTd taggersFor: streamBuffer) ].

Transcript cr; show: 'Data File: ',fname.
Transcript cr.

taggers do: [ :aTagger |
    Transcript cr; show: ' Tagger Number: ',
       (aTagger id printPaddedTo: 2 with: $ ),
       '  Tagging Time: ',(( Time millisecondsToRun: [
           aTagger assignTags ])
               printPaddedTo: 6 with: $ ) ].

streamBuffer := streamBuffer asOrderedCollection.

temp := (streamBuffer select: [ :aPulse | aPulse isUnclaimed ]) size.
Transcript cr; show: '     Unclaimed: ', (temp printPaddedTo: 6 with: $ ).

temp := (streamBuffer select: [ :aPulse |
    (aPulse isUnclaimed or: [ aPulse isMixed]) not ]) size.
Transcript cr; show: ' Singly Tagged: ', (temp printPaddedTo: 6 with: $ ).

temp := (streamBuffer select: [ :aPulse | aPulse isMixed ]) size.
Transcript cr; show: '     Contested: ', (temp printPaddedTo: 6 with: $ ).

temp := (streamBuffer select: [ :aPulse |
    (aPulse tagCollection select: [ :aTagId |
        aTagId > 9 ]) isEmpty not ]) size.
Transcript cr; show: 'Reverse Tagged: ', (temp printPaddedTo: 6 with: $ ).

Transcript cr; cr; show: '          Total: ', (streamBuffer
                                    size printPaddedTo: 6 with: $ ).
Transcript cr; cr.

tds do: [ :aTd |
    Transcript show: aTd printString; cr ].

histograms := OrderedCollection new.

taggers do: [ :aTagger |
    histograms add: (NumericHistogram binWidth: 1).
    count := 0.
    aTagger monitor keysDo: [ :aKey |
        histograms last add: (self monitorConstants at: aKey) x
                        withOccurrences: (aTagger monitor at: aKey).
        count := count + (aTagger monitor at: aKey) ].
    Transcript cr; show: 'Total number of state transitions for ',
                         aTagger id printString, ': ',
                         count printString ].
histograms inspect.

^ taggers.
```

```
! !

!MonitoredSequenceStreamTagger publicMethods !

assignTags
"
    Process control method.

    If called the first time, initialize the streamReference and
    currentIndex to the oldest pulse in the streamBuffer.

    Iterate over the streamBuffer.  Determine the delta TOA between the
    streamReference pulse and the current pulse.  Execute the part of the
    tagging algorithm defined by state. Once the existing pulses in the
    streamBuffer have been processed, exit so that more pulses can be added
    to the stream.

        MonitoredSequenceStreamTagger testOn: PulseStream open for: 1000.
        SequenceTagger testOn: PulseStream open for: 1000.
        MonitoredSequenceStreamTagger testPulseByPulseOn: PulseStream open for:
1000.
"

    streamReference isNil
        ifTrue: [
            streamReference := streamBuffer firstIndex.
            (streamBuffer at: streamReference) reference.
            currentIndex := streamBuffer nextIndexFrom: streamReference ].

    [ streamBuffer contains: currentIndex ] whileTrue: [

        self computeDelta.
        self executeState.

        currentIndex := streamBuffer nextIndexFrom: currentIndex ].
!

id
"
    Answer the tag id of this tagger.
"
    ^id
!

monitor
"
"
    ^monitor.
!

streamBuffer: aStreamBuffer
"
    Set the streamBuffer to aStreamBuffer.
"
    streamBuffer := aStreamBuffer.
!

updateWith: aTaggerDescriptor
"
    Update a tagger with aTaggerDescriptor as the new set of parameters.
    For any nil parameter, leave the corresponding paramater in the tagger
    unchanged.

    A change of any interval in either the rf or priIntervals collection
```

```
    requires that the entire collection be present in the descriptor.
"
    | priIntervals minT sumOfDeltas |

    aTaggerDescriptor priIntervals isEmpty
        ifFalse: [
            priIntervals := aTaggerDescriptor priIntervals.
            numberOfIntervals := priIntervals size.

            priLeft := Array new: numberOfIntervals.
            priRight := Array new: numberOfIntervals.
            maxT := 0.
            groupIntervalLeft := 0.
            groupIntervalRight := 0.

            sumOfDeltas := 0.
            1 to: numberOfIntervals do: [ :index |
                priLeft at: index put: (priIntervals at: index) left.
                priRight at: index put: (priIntervals at: index) right.
                maxT := maxT max: (priIntervals at: index) right.
                groupIntervalLeft :=
                    groupIntervalLeft + (priIntervals at: index) left.
                groupIntervalRight :=
                    groupIntervalRight + (priIntervals at: index) right.
                minT = nil
                    ifTrue: [ minT := (priIntervals at: index) left ].
                minT := minT min: (priIntervals at: index) left.
                sumOfDeltas := sumOfDeltas + (priIntervals at: index) width ]].

    aTaggerDescriptor missingPulseThreshold isNil
        ifFalse: [ missingPulseThreshold :=
                        aTaggerDescriptor missingPulseThreshold ].
    aTaggerDescriptor minMatchCount isNil
        ifFalse: [
            minMatchCount := aTaggerDescriptor minMatchCount.
            matchedPulses := Array new: minMatchCount ].
! !

!MonitoredSequenceStreamTagger privateMethods !

accumulateTimeFor: aKey
"
"
    monitor at: aKey ifAbsent: [ self error: 'monitor key missing' ].

    monitor at: aKey put:
        ( (monitor at: aKey) + ((self class monitorConstants at: aKey) y) ).
!

accumulateTimeForMatchedPulsesLoop
"
"
    self accumulateTimeFor: #accumulateTimeForMatchedPulsesLoop.
!

accumulateTimeForNumberOfIntervalsLoop
"
"
    self accumulateTimeFor: #accumulateTimeForNumberOfIntervalsLoop.
!

addNextInterval
"
    Increment the phase of the interval set.  Add the next left and right
```

```
    intervals to the existing left and right interval values.
"

    (phase := phase + 1) > numberOfIntervals ifTrue: [ phase := 1 ].
    left  := left  + (priLeft  at: phase).
    right := right + (priRight at: phase).

    self accumulateTimeFor: #addNextInterval.
!

computeDelta
"
"
    delta := (streamBuffer at: currentIndex)
                    toaDifferenceAsus: (streamBuffer at: streamReference).

    self accumulateTimeFor: #computeDelta.
!

deltaGreaterThanMaxT
"
"
    self accumulateTimeFor: #deltaGreaterThanMaxT.

    ^delta > maxT.
!

deltaGreaterThanRight
"
"
    self accumulateTimeFor: #deltaGreaterThanRight.

    ^delta > right.
!

deltaLessThanLeft
"
"
    self accumulateTimeFor: #deltaLessThanLeft.

    ^delta < left.
!

deltaMatches
"
"
    self accumulateTimeFor: #deltaMatches.

    ^( delta >= left and: [ delta <= right ] )
!

executeState
"
    Execute a method corresponding to the symbol in state.  The states can be
    #sync, #matchLock, #lockon or #skipPulse.
"
    self accumulateTimeFor: #executeState.

    state = #sync
        ifTrue: [
            self sync.
            ^self ].
    state = #matchLock
        ifTrue:  [
```

```
                self matchLock.
                ^self ].
        state = #lockon
            ifTrue: [
                self match.
                ^self ].
         state = #skipPulse
            ifTrue: [
                self skipPulse.
                ^self ].
!

incrementMatchCount
"
"
    self accumulateTimeFor: #incrementMatchCount.

    matchCount := matchCount + 1.
!

incrementSkipCount
"
"
    self accumulateTimeFor: #incrementSkipCount.

    skipCount := skipCount + 1.
!

initializeMatchCount
"
"
    self accumulateTimeFor: #initializeMatchCount.

    matchCount := 1.
!

initializeMonitor
"
"
    monitor := Dictionary new.

    self class monitorConstants keysDo: [ :aKey |
        monitor at: aKey put: 0 ].
!

initializeOn: aTaggerDescriptor for: aStreamBuffer
"
    Initialize the tagger on aTaggerDescriptor for a streamBuffer.  The
    streamBuffer is unique for each coarse channel.
"
    | priIntervals minT sumOfDeltas |

    self initializeMonitor.

    priIntervals := aTaggerDescriptor priIntervals.
    numberOfIntervals := priIntervals size.

    id := aTaggerDescriptor tagId.
    priLeft := Array new: numberOfIntervals.
    priRight := Array new: numberOfIntervals.
    maxT := 0.
    groupIntervalLeft := 0.
    groupIntervalRight := 0.
```

```
    sumOfDeltas := 0.
    1 to: numberOfIntervals do: [ :index |
        priLeft at: index put: (priIntervals at: index) left.
        priRight at: index put: (priIntervals at: index) right.
        maxT := maxT max: (priIntervals at: index) right.
        groupIntervalLeft :=
            groupIntervalLeft + (priIntervals at: index) left.
        groupIntervalRight :=
            groupIntervalRight + (priIntervals at: index) right.
        minT = nil
            ifTrue: [ minT := (priIntervals at: index) left ].
        minT := minT min: (priIntervals at: index) left.
        sumOfDeltas := sumOfDeltas + (priIntervals at: index) width ].

    aTaggerDescriptor missingPulseThreshold isNil
        ifTrue: [
           sumOfDeltas = 0
                ifTrue: [
                    missingPulseThreshold :=
                            self class defaultMaxMissingPulses ]
                ifFalse: [
                    missingPulseThreshold :=
                        (self class defaultMaxMissingPulses min: (
                            (minT // sumOfDeltas) * numberOfIntervals))
                                max: 2 ] ]
        ifFalse: [
            missingPulseThreshold :=
                aTaggerDescriptor missingPulseThreshold ].

    aTaggerDescriptor minMatchCount isNil
        ifTrue: [
            minMatchCount := numberOfIntervals -
                aTaggerDescriptor nbrUniquePriIntervals +
                                        self class defaultMinMatchCount ]
        ifFalse: [
            minMatchCount := aTaggerDescriptor minMatchCount ].

    matchedPulses := Array new: minMatchCount.

    streamBuffer := aStreamBuffer.
    state := #sync.
!

match
"
    If the state is #match, this method is called for each iteration of the
    streamBuffer.  Begin by comparing the delta TOA to the current predicted
    interval bounded by left and right.  If the compare is successful, tag
    the pulse associated with the currentIndex, set the streamReference
    to the currentIndex and find the next interval in the sequence.

    If the compare is not sucessful and if delta is less than the left
    predicted interval, exit allowing the next pulse in the stream to be
    processed.  This condition occurs when a pulse from another emitter is
    encountered in a TOA position which is between the pulses which this
    tagger is tagging.

    If this second test fails, there is at least one missing pulse.  If the
    missingPulseThreshold is zero (no missing pulses are allowed), switch
    states to acquisition and exit (lockon is lost).

    If one or more missing pulses are allowed, add the next predicted interval
    to the current interval, set up for missing pulse processing by setting
    the state to #skipPulses, move the currentIndex back one (it will be
```

```
    advanced one by the control method) and exit.  The control method will
    then force the execution of the skipPulse algorithm as many times as
    required to match the current pulse or lose lockon.
"
    self accumulateTimeFor: #match.

    self deltaMatches
        ifTrue:  [
            self tagAt: currentIndex.
            self referenceCurrentPulse.
            self setNextInterval.
            ^self ].
    self deltaLessThanLeft
        ifTrue: [ ^self ].

    self missingPulseThresholdIsZero
        ifTrue: [
            self switchToAcquisition.
            ^self ].

    self addNextInterval.
    self setupToSkip.
!

matchCountGreaterThanMinMatchCount
"
"
    self accumulateTimeFor: #matchCountGreaterThanMinMatchCount.

    ^ matchCount >= minMatchCount
!

matchLock
"
    Determine if delta lies within the predicted interval.  If true, determine
    if the total number of matches so far meets the minimumMatchCount
    thershold.  If this is the case, switch to the lockon mode and exit.

    If either of the above tests fail and if the delta is greater than the
    predicted right interval, the lockon attempt has failed.  Move the stream
    reference to the next pulse and set the state to #sync to restart the
    search for lockon.

    If all of the above tests fail, remain in the #matchLock state so that
    the next interval in the pulse stream can be tested.
"

    self accumulateTimeFor: #matchLock.

    self deltaMatches
        ifTrue: [
            self sequenceMatches
                ifTrue: [
                    self setNextInterval.
                    self switchToLockon.
                    ^ self ].
            self addNextInterval.
            ^self ].
    self deltaGreaterThanRight
        ifTrue: [
            self referenceNextPulse.
            self setStateToSync ].
!
```

```
missingPulseThresholdIsZero
"
"
    self accumulateTimeFor: #missingPulseThresholdIsZero.

    ^ missingPulseThreshold = 0
!

referenceCurrentPulse
"
    Dereference the current streamReference pulse, set the streamReference
    to the currentIndex and establish a reference on that pulse.  A reference
    count is maintained by the streamBuffer to establish referencing.
"
    (streamBuffer at: streamReference) deReference.
    streamReference := currentIndex.
    (streamBuffer at: streamReference) reference.

    self accumulateTimeFor: #referenceCurrentPulse.
!

referenceNextPulse
"
    Dereference the current streamReference pulse, set the streamReference
    to the next pulse and establish a reference on that pulse.  A reference
    count is maintained by the streamBuffer to establish referencing.  Set
    the currentIndex to the streamReference.
"
    (streamBuffer at: streamReference) deReference.
    streamReference := streamBuffer nextIndexFrom: streamReference.
    (streamBuffer at: streamReference) reference.
    currentIndex := streamReference.

    self accumulateTimeFor: #referenceNextPulse.
!

sequenceMatches
"
    Answer true if the stream fragment matches the currect number of intervals
    in the known PRI sequence.  This method is entered if a TOA interval
    matches the current interval.

    Store the index of the current pulse in an array.  If the matchCount
    meets the minMatchCount, tag the streamReferenced pulse, tag all other
    pulses which matched the sequence, and return true.

    If the minMatchCount has not been exceeded, increment the matchCount
    and return false.
"

    self accumulateTimeFor: #sequenceMatches.

    self storeCurrentIndexInMatchList.

    self matchCountGreaterThanMinMatchCount
        ifTrue: [
            self tagAt: streamReference.
            1 to: matchCount do: [ :pulseIndex |
                self accumulateTimeForMatchedPulsesLoop.
                self tagAt: (matchedPulses at: pulseIndex) ].
            ^true ].
    self incrementMatchCount.
    ^false.
!
```

```
setNextInterval
"
    Increment the phase of the interval set.  Set the existing right
    and left intervals to the next predicted interval.
"

    (phase := phase + 1) > numberOfIntervals ifTrue: [ phase := 1 ].
    left  := priLeft  at: phase.
    right := priRight at: phase.

    self accumulateTimeFor: #setNextInterval.
!

setStateToMatchLock
"
"
    self accumulateTimeFor: #setStateToMatchLock.

    state := #matchLock.
!

setStateToSync
"
"
    self accumulateTimeFor: #setStateToSync.

    state := #sync.
!

setupPhaseFor: priIndex
"
"
    self accumulateTimeFor: #setupPhase.

    phase := priIndex.
    left  := priLeft  at: phase.
    right := priRight at: phase.
!

setupToSkip
"
"
    state := #skipPulse.
    skipCount := 0.
    currentIndex := streamBuffer previousIndexFrom: currentIndex.

    self accumulateTimeFor: #setupToSkip.
!

skipCountLessThanMissingPulseThreshold
"
"
    self accumulateTimeFor: #skipCountLessThanMissingPulseThreshold.

    ^ skipCount < missingPulseThreshold
!

skipPulse
"
    Execute if a missing pulse has been detected during lockon. While the
    skipCount does not exceed the missingPulseThreshold, test the current
    TOA interval against each additional predicted interval in the sequence.
    If a match can be found, tag the current pulse, set the predicted interval
```

```
    to the next interval in the sequence, switch the state back to #lockon
    and exit.

    If the current TOA interval is less than a current predicted interval,
    exit so that the next pulse can be obtained.  This occurs when a pulse
    from another emitter is interleaved.

    If the missingPulseThreshold is exceeded, switch to acquisition mode.
"
    self accumulateTimeFor: #skipPulse.

    [ self skipCountLessThanMissingPulseThreshold ]
        whileTrue: [

            self deltaMatches
                ifTrue: [
                    self tagAt: currentIndex.
                    self setNextInterval.
                    self switchToLockon.
                    ^self ].
            self deltaLessThanLeft
                ifTrue: [ ^self ].

            self addNextInterval.
            self incrementSkipCount ].

    self switchToAcquisition.
!

storeCurrentIndexInMatchList
"
"
    self accumulateTimeFor: #storeCurrentIndexInMatchList.

    matchedPulses at: matchCount put: currentIndex.
!

switchToAcquisition
"
    Set the streamReference to the next pulse in the stream.  Set the state
    to @sync.
"
    self referenceNextPulse.
    state := #sync.

    self accumulateTimeFor: #switchToAcquisition.
!

switchToLockon
"
    Set the streamReference to the currentIndex and switch the state to
    #lockon.
"
    self referenceCurrentPulse.
    state := #lockon.

    self accumulateTimeFor: #switchToLockon.
!

sync
"
    Sync is the initial phase of acquisition.  If the curent delta is greater
    than maxT ( the largest right value of all of the intervals in the known
    sequence ), set the streamReference to the next pulse ( the first interval
```

```
    is not one in the seqence ).

    If the current interval matches one of the intervals in the known sequence,
    set the phase to the index of that interval, set the matchCount to one
    and determine if the sequence matches (if the matchCount meets the
    minMatchCount requirement).

    If the sequence matches, determine the next prediction interval from
    the sequence and switch to lockon.

    If the sequence does not match, get the next prediction window and
    switch to the matchLock state where subsequent pulses will be matched.

    If none of the intervals match, exit so that the interval associated with
    the next pulse can be tested.
"
    self accumulateTimeFor: #sync.

    self deltaGreaterThanMaxT
        ifTrue: [
            self referenceNextPulse.
            ^self ].

    1 to: numberOfIntervals do: [ :priIndex |
        self accumulateTimeForNumberOfIntervalsLoop.
        self setupPhaseFor: priIndex.

        self deltaMatches
            ifTrue: [
                self initializeMatchCount.
                self sequenceMatches
                    ifTrue: [
                        self setNextInterval.
                        self switchToLockon.
                        ^self ].

                self addNextInterval.
                self setStateToMatchLock.
                ^ self ] ].
!

tagAt: anIndex
"
"
    streamBuffer tagWith: id at: anIndex.

    self accumulateTimeFor: #tagAt.
! !

MonitoredSequenceStreamTagger initialize
!
```

P148505.PDF [Page: 83 of 108]

Appendix C

Smalltalk State Machine Model

Experimental Results

```
Data File: e015.dta

 Tagger Number:  1  Tagging Time:   3350
 Tagger Number:  2  Tagging Time:   3950

     Unclaimed:    103
 Singly Tagged:    837
     Contested:     60
Reverse Tagged:      0

         Total:   1000

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 1
 priIntervals: (1174 To 1182) (873 To 881) (958 To 966) (827 To 834)
 misPt: 4 minMth: 1

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 2
 priIntervals: (958 To 966)
 misPt: 4 minMth: 2


Total number of state transitions for 1: 4921
Total number of state transitions for 2: 4914
```

P148505.PDF [Page: 85 of 108]


```
Data File: e069.dta

 Tagger Number:  1  Tagging Time:   3520

     Unclaimed:       7
 Singly Tagged:     993
     Contested:       0
Reverse Tagged:       0

         Total:    1000

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 1
 priIntervals: (2220 To 2372)
 misPt: 20 minMth: 1


Total number of state transitions for 1: 5469
```

P148505.PDF [Page: 86 of 108]


```
Data File: e104.dta

 Tagger Number:  1  Tagging Time:   3680
     Unclaimed:    132
  Singly Tagged:   868
     Contested:      0
Reverse Tagged:      0

          Total:  1000

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 1
 priIntervals: (953 To 1037)
 misPt: 10 minMth: 1


Total number of state transitions for 1: 4851
```

P148505.PDF [Page: 87 of 108]


```
Data File: e126.dta

 Tagger Number:  1  Tagging Time:   2850
 Tagger Number:  2  Tagging Time:   5550
 Tagger Number:  3  Tagging Time:   3570
 Tagger Number:  4  Tagging Time:   7080
 Tagger Number:  5  Tagging Time:   2200
 Tagger Number:  6  Tagging Time:   7200
 Tagger Number:  7  Tagging Time:   3240
     Unclaimed:     16
 Singly Tagged:    814
    Contested:     170
Reverse Tagged:      0

         Total:   1000

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 1
 priIntervals: (1203 To 1215)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 2
 priIntervals: (1188 To 1195)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 3
 priIntervals: (539 To 551) (572 To 580)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 4
 priIntervals: (1262 To 1269)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 5
 priIntervals: (298 To 309)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 6
 priIntervals: (1319 To 1326)
 misPt: 10 minMth: 2

TaggerDescriptor of: MonitoredSequenceStreamTagger
 tagId: 7
 priIntervals: (540 To 569)
 misPt: 10 minMth: 2


Total number of state transitions for 1: 3313
Total number of state transitions for 2: 6858
Total number of state transitions for 3: 4725
Total number of state transitions for 4: 9080
Total number of state transitions for 5: 3045
Total number of state transitions for 6: 8880
Total number of state transitions for 7: 3863
```

P148505.PDF [Page: 88 of 108]

Appendix D

VHDL SOURCE CODE LISTINGS

P148505.PDF [Page: 89 of 108]

Appendix D VHDL SOURCE CODE LISTINGS

File : pulseStream.hdl
Description : Behavioral Model for the Pulse Stream functions required to support the tagger performance evaluation.

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;
USE STD.TEXTIO.ALL;

ENTITY pulseStream IS
        PORT (
                reset     : IN qsim_state;
                clk       : IN qsim_state;
                ready     : OUT qsim_state;
                tagIdx    : IN qsim_state_vector(7 DOWNTO 0);
                refIdx    : IN qsim_state_vector (9 DOWNTO 0);
                curIdx    : IN qsim_state_vector (9 DOWNTO 0);
                tagRef    : IN qsim_state;
                tagCur    : IN qsim_state;
                tagMatch : IN qsim_state;

                MatchIdx          : IN qsim_state_vector (9 DOWNTO 0);
                curIdx_OutOfRange       : OUT qsim_state;
                outData1 : OUT qsim_state_vector (31 DOWNTO 0);
                outData2 : OUT qsim_state_vector (31 DOWNTO 0);
                delta : OUT qsim_state_vector (31 DOWNTO 0);

                incRefCnt         : IN qsim_state;
                decRefCnt         : IN qsim_state;
                clrRefCnt         : IN qsim_state;

                refCnt    : OUT qsim_state_vector (2 DOWNTO 0);
                tagOut1   : OUT qsim_state_vector (7 DOWNTO 0);
                tagOut2   : OUT qsim_state_vector (7 DOWNTO 0));
END;

ARCHITECTURE behavioral OF pulseStream IS

CONSTANT          RAMSIZE : positive := 1100;

TYPE dataType IS ARRAY (natural RANGE <>) OF INTEGER;
TYPE tagType IS ARRAY (natural RANGE <>) OF INTEGER;
TYPE refCntType IS ARRAY (natural RANGE <>) OF INTEGER;

FILE              dataFile        : TEXT IS IN "e015.dta";
SIGNAL            ramData         : dataType (0 TO RAMSIZE);
SIGNAL            tagData           : tagType (0 TO RAMSIZE);
SIGNAL            refCntData        : refCntType (0 TO RAMSIZE);
```

P148505.PDF [Page: 90 of 108]


```
SIGNAL          tmpBusy          : qsim_state;
SIGNAL          tmpCurIdx : integer;

BEGIN

        outData1 <=  to_qsim_state(ramData(to_integer(('0' & refIdx),0)),32);
        outData2 <=  to_qsim_state(ramData(to_integer(('0' & curIdx),0)),32);
        tagOut1 <=  to_qsim_state(tagData(to_integer(('0' & refIdx),0)),8);
        tagOut2 <=  to_qsim_state(tagData(to_integer(('0' & curIdx),0)),8);
        refCnt <= to_qsim_state(refCntData(to_integer(('0' & refIdx),0)),3);
--      tmpCurIdx <= to_integer(curIdx,0);
        delta <=  to_qsim_state( ramData(to_integer(('0' & curIdx),0)) -
                                 ramData(to_integer(('0' & refIdx),0)),32);
-- busy <= tmpBusy;

        initialize : PROCESS

                VARIABLE    data     : LINE;
                VARIABLE    good     : BOOLEAN;
                VARIABLE    toa      : INTEGER;
                VARIABLE    temp     : INTEGER;
                VARIABLE    index    : INTEGER;

                BEGIN
                        WAIT ON reset,clk;

                        IF (reset = '0' AND reset'EVENT) THEN
                                ASSERT FALSE
                                        REPORT "Initializing FIFO ptrs"
                                        SEVERITY NOTE;
                                index := 0;
                                WHILE NOT ENDFILE(dataFile) LOOP
                                        READLINE(dataFile,data);
                                        READ(data,toa,good);
                                        ramData(index) <= toa;
                                        index := index + 1;
                                        IF (index > RAMSIZE) THEN
                                                ASSERT FALSE
                                                        REPORT "RAM FULL - data lost"
                                                        SEVERITY NOTE;
                                                index := index - 1;
                                        END IF;
                                END LOOP;
                                FOR i IN 1 TO RAMSIZE LOOP
                                        tagData(i-1) <= 0;
                                        refCntData(i-1) <= 0;
                                END LOOP;
                                ready <= '1';
```

P148505.PDF [Page: 91 of 108]


```
END IF;

IF (clk'EVENT AND clk'LAST_VALUE = '0' AND clk = '1') THEN
        IF (incRefCnt = '0' AND decRefCnt = '0') THEN
                tmpBusy <= '0';
        END IF;
        IF (((to_integer(('0' & refIdx),0)) < index) AND (tagRef = '1'))
        THEN
                ASSERT FALSE
                REPORT "Tagging Pulse "
                SEVERITY NOTE;
        tagData(to_integer(('0' & refIdx),0)) <=
                to_integer(to_qsim_state(tagData(to_integer(('0' &
                refIdx),0))),8) OR tagIdx,0);
        IF (tmpBusy = '0') THEN
                IF (incRefCnt = '1') THEN
                        tmpBusy <= '1';
                        refCntData(to_integer(('0' & refIdx),0)) <=
                                refCntData(to_integer(('0' & refIdx),0)) +
                                1;
                ELSIF (decRefCnt = '1') THEN
                        tmpBusy <= '1';
                        refCntData(to_integer(('0' & refIdx),0)) <=
                                refCntData(to_integer(('0' & refIdx),0)) -
                                1;
                END IF;
        END IF;
END IF;

IF (((to_integer(('0' & curIdx),0)) < index) AND (tagCur = '1')) THEN
        ASSERT FALSE
                REPORT "Tagging Pulse "
                SEVERITY NOTE;
        tagData(to_integer(('0' & curIdx),0)) <=
                        to_integer(to_qsim_state(tagData(to_integer(('0' &
                        curIdx),0))),8) OR tagIdx,0);
END IF;
IF ((to_integer(('0' & curIdx),0)) < index)  THEN
        curIdx_OutOfRange <= '0';
ELSE
        curIdx_OutOfRange <= '1';
END IF;
IF (((to_integer(('0' & matchIdx),0)) < index) AND (tagMatch = '1'))
THEN
        ASSERT FALSE
                REPORT "Tagging Pulse "
                SEVERITY NOTE;
        tagData(to_integer(('0' & matchIdx),0)) <=
                        to_integer(to_qsim_state(tagData(to_integer(('0' &
```

P148505.PDF [Page: 92 of 108]


```
                                                    matchIdx),0)),8) OR tagIdx,0);
                        END IF;
                    END IF;
                END PROCESS;

END behavioral;
```

P148505.PDF [Page: 93 of 108]

File: tagger.hdl
Description: Register Transfer Level Model for the Sequence Tagger State Machine

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;


ENTITY tagger IS
        PORT (
                reset           : IN qsim_state;
                clk             : IN qsim_state;

                assignTags      : IN qsim_state;
                taggingDone     : OUT qsim_state;

                tagRef,tagCur,tagMatch   : OUT qsim_state;
                incRefCnt,decRefCnt      : OUT qsim_state;
                numIntervals             : IN qsim_state_vector(3 DOWNTO 0);
                refIdxOut,curIdxOut,matchIdxOut : OUT qsim_state_vector(9 DOWNTO 0);

                curIdx_OutOfRange : IN QSIM_STATE;
                matchCnt_GT_MinMatchCnt : IN QSIM_STATE;
                matchCnt_EQ_0           : IN QSIM_STATE;
                clrMatchCnt,inc_ndec,enMatchCnt : OUT qsim_state;
                clrPhase,incPhase : OUT qsim_state;
                reload, addInterval       : OUT qsim_state;

                phaseOut                  : OUT qsim_state_vector(3 DOWNTO 0);
                phase_LT_numIntervals     : IN qsim_state;

                MissingPulseThresh        : IN qsim_state_vector(4 DOWNTO 0);

                delta_GT_maxT, delta_GE_left, delta_LE_right : qsim_state
        );
END;

ARCHITECTURE rtl OF tagger IS

TYPE smStates IS (      idle, assign1, sync, syncSeqMatches, matchLock, syncTagAllMatches,
                        matchLockSequenceMatches, matchLockTagAll, match, skip, done);

SIGNAL refIdx,curIdx,matchIdx : QSIM_STATE_VECTOR(9 DOWNTO 0);
SIGNAL phase : QSIM_STATE_VECTOR(3 DOWNTO 0);
SIGNAL smState : smStates := idle;
SIGNAL skipCnt : qsim_state_vector(4 downto 0);

BEGIN
```

P148505.PDF [Page: 94 of 108]


```
refIdxOut <= refIdx;
curIdxOut <= curIdx;
phaseOut <= phase;


main : PROCESS  (reset, clk)

VARIABLE      refPlusOne : INTEGER;

BEGIN

        IF (reset = '0') THEN
                smState <= idle;
                tagState <= sync;
        ELSIF  (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN

                clrMatchCnt <= '1';
                enMatchCnt <= '1';
                clrPhase <= '0';
                incPhase <= '0';
                incRefCnt <= '0';
                decRefCnt <= '0';
                reload <= '0';
                addInterval <= '0';

                IF (smState = idle) THEN
                        taggingDone <= '1';
                ELSE
                        taggingDone <= '0';
                END IF;


                CASE smState IS
                        WHEN idle =>
                                IF (assignTags = '1') THEN
                                        smState <= assign1;
                                        refIdx <= "0000000000";
                                        incRefCnt  <= '1';
                                        curIdx <= "0000000000";
                                END IF;

                        WHEN assign1 =>
                                curIdx <= curIdx + "0000000001";
                                IF (curIdx_OutOfRange = '1') THEN
                                        smState <= done;
                                ELSE
                                        clrMatchCnt <= '0';
                                        reload <= '1';
                                        phase <= "0000";
```

P148505.PDF [Page: 95 of 108]


```
                smState <= sync;
        END IF;

WHEN sync =>
        IF (delta_GT_maxT = '1') THEN
                decRefCnt <= '1';
                refIdx <= refIdx + "0000000001";
                curIdx <= refIdx + "0000000001";
                phase <= "0000";
                smState <= assign1;
        ELSIF ((delta_GE_left = '1') AND (delta_LE_right = '1'))
        THEN
                curIdx <= curIdx + "0000000001";
                IF (curIdx_OutOfRange = '1') THEN
                        smState <= done;
                ELSE
                        smState <= syncSeqMatches;
                END IF;
                inc_ndec <= '1';
                enMatchCnt <= '0';
                addInterval <= '1';
        ELSIF (phase = numIntervals) THEN
                smState <= assign1;
                phase <= "0000";
        ELSE
                reload <= '1';
        END IF;
        phase <= phase + "0001";


WHEN syncSeqMatches =>
        IF (matchCnt_GT_MinMatchCnt = '1') THEN
                tagRef <= '1';
                tagCur <= '1';
                smState <= syncTagAllMatches;
        ELSE
                smState <= matchLock;
        END IF;


WHEN syncTagAllMatches =>
        IF (matchCnt_EQ_0 = '1') THEN
                refIdx <= curIdx - "0000000001";
                reload <= '1';
                smState <= match;
        ELSE
                tagMatch <= '1';
                inc_ndec <= '0';
                enMatchCnt <= '0';
```

P148505.PDF [Page: 96 of 108]


```
            END IF;

        WHEN matchLock =>
            IF ((delta_GE_left = '1') AND (delta_LE_right = '1'))
            THEN
                smState <= matchLockSequenceMatches;
            ELSE
                smState <= sync;
            END IF;
            phase <= phase + "0001";
            IF (phase >= numIntervals) THEN
                phase <= "0000";
            END IF;

        WHEN matchLockSequenceMatches =>
            IF (matchCnt_GT_MinMatchCnt = '1') THEN
                tagRef <= '1';
                tagCur <= '1';
                smState <= matchLockTagAll;
            ELSE
                smState <= matchLock;
            END IF;

        WHEN matchLockTagAll =>
            smState <= match;

        WHEN match =>
            IF ((delta_GE_left = '1') AND (delta_LE_right = '1'))
            THEN
                refIdx <= curIdx;
                    curIdx <= curIdx + "0000000001";
                reload <= '1';
                phase <= phase + "0001";
                IF (phase >= numIntervals) THEN
                    phase <= "0000";
                END IF;

                    IF (curIdx_OutOfRange = '1') THEN
                        smState <= done;
                    END IF;
            ELSIF (delta_GE_left = '0') THEN
                curIdx <= curIdx + "0000000001";
            ELSE
                skipCnt <= "00000";
```

P148505.PDF [Page: 97 of 103]


```
                                    smState <= skip;
                                    phase <= phase + "0001";
                                    IF (phase >= numIntervals) THEN
                                        phase <= "0000";
                                    END IF;

                                    addInterval <= '1';
                                END IF;


                            WHEN skip =>
                                IF (skipCnt < MissingPulseThresh) THEN
                                    skipCnt <= skipCnt + "00001";
                                    IF (delta_GE_left = '0') THEN
                                        curIdx <= curIdx + "0000000001";
                                    ELSIF ((delta_GE_left = '1') AND
                                    (delta_LE_right = '1')) THEN
                                        phase <= phase + "0001";
                                        IF (phase >= numIntervals) THEN
                                            phase <= "0000";
                                        END IF;
                                            refIdx <= curIdx;
                                            curIdx <= curIdx +
                                            "0000000001";
                                            reload <= '1';
                                            IF (curIdx_OutOfRange = '1')
                                            THEN
                                                smState <= done;
                                    ELSE
                                        smState <= match;
                                            END IF;
                                END IF;
                            ELSE
                                smState <= done;
                            END IF;


                        WHEN done =>
                            IF (curIdx_OutOfRange = '1') THEN
                                smState <= idle;
                            ELSE
                                smState <= assign1;
                            END IF;
                    END CASE;
                END IF;
            END PROCESS main;

        END;
```

P148505.PDF [Page: 98 of 108]

File : **alu32.hdl**
Description: **Autologic blocks genertaed VHDL model for 32 bit ALU.**

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

--

-- alu32 Entity Description
entity alu32 is
  port(
            clk     : in qsim_state;
            add : in qsim_state;
            reload : in qsim_state;
                  A: in qsim_state_vector(31 downto 0);
                  D: out qsim_state_vector(31 downto 0)
  );
end alu32;

-- alu32 Architecture Description
architecture rtl of alu32 is
            signal pre_D : qsim_state_vector(32 downto 0);
            signal accum : qsim_state_vector(32 DOWNTO 0);

begin


  loadAcc : process(A,clk)
            variable fct_out : qsim_state_vector(32 downto 0);
            variable a_ext,b_ext : qsim_state_vector(32 downto 0);
            variable carry_ext : qsim_state_vector(1 downto 0);
            variable msb : integer;
  begin

            a_ext := '0' & A;
            IF (reload = '1') THEN
                  fct_out := '0' & A;
            ELSE
                  fct_out := accum;
            END IF;
            pre_D <= fct_out;
  end process loadAcc;

  ARITHMETIC_Process: process(clk)
            variable fct_out : qsim_state_vector(32 downto 0);
            variable a_ext,b_ext : qsim_state_vector(32 downto 0);
             variable carry_ext : qsim_state_vector(1 downto 0);
            variable msb : integer;
  begin
```

P148505.PDF [Page: 99 of 108]


```
            IF  (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN

                a_ext := '0' & A;

                IF (add = '1') THEN
                        -- zero extend inputs to include carry bit
                        accum <= pre_D + a_ext;
                ELSIF (reload = '1') THEN
                        accum <= pre_D;
                END IF;
            END IF;

  end process ARITHMETIC_Process;



  -- Assign the outputs
  D <= pre_D(31 downto 0);

end rtl;
```

P148505.PDF [Page: 100 of 108]

File: **reg32.hdl**
Description: **16 deep x 32 bit wide regiter file generated by Autologic blocks**

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

--
-- Written by LL_to_VHDL at Fri Feb  4 01:37:38 1994
-- Parameterized Generator Specification to VHDL Code
--
--
-- LogicLib generator called: REGISTER_FILE
-- Passed Parameters are:
--   tinst name = reg
--   parameters are:
--      W = 32
--      D = 16
--      read_ports = 1
--

-- reg Entity Description
entity reg is
   port(
      DIN1: in qsim_state_vector(31 downto 0);
      RADR1: in qsim_state_vector(3 downto 0);
      WADR1: in qsim_state_vector(3 downto 0);
      DOUT1: out qsim_state_vector(31 downto 0);
      LD1,WE1: in qsim_state
   );
end reg;

-- reg Architecture Description
architecture rtl of reg is
   subtype ramword is qsim_state_vector (31 downto 0);
   type rammemory is array (15 downto 0) of ramword;
   signal ram : rammemory;
begin

   REGISTER_FILE_read_Process: process(ram,RADR1)
      variable raddr1 : integer range 0 to 15;
   begin
      -- Process the read port number 1
      -- convert address to integer
      raddr1 := to_integer('0' & RADR1,0);
      DOUT1  <= ram(raddr1);
   end process REGISTER_FILE_read_Process;

   REGISTER_FILE_write_Process: process(DIN1,WE1,LD1,WADR1)
      variable waddr : integer range 0 to 15;
```

P148505.PDF [Page: 101 of 108]


```
    variable load : qsim_state;
  begin
    -- write mode? (need both WE1 and LD1 to be high)

    -- convert address to integer
    waddr := to_integer('0' & WADR1,0);
    load  := LD1 and WE1;
    if (load = '1') then
      ram(waddr) <= DIN1;
    end if;
  end process REGISTER_FILE_write_Process;
end rtl;
```

P148505.PDF [Page: 102 of 108]

Appendix E

MENTOR QUICKSIM II

START-UP SHELL SCRIPT

P148505.PDF [Page: 103 of 108]

Appendix E **MENTOR QUICKSIM II START-UP SHELL SCRIPT**

File : **FORCES.DATA**
Description : **QuickSim II "dofile" - sets up trace display and tagger data for e015 sequence tagger.**

```
$add_traces(@nooverlay, void, void, "/clk");
$add_traces(@nooverlay, void, void, "/reset");
$add_traces(@nooverlay, void, void, "/delta_le_right");
$add_traces(@nooverlay, void, void, "/delta_ge_left");
$add_traces(@nooverlay, void, void, "/delta_gt_maxt");
$add_traces(@nooverlay, void, void, "/left[31:0]");
$add_traces(@nooverlay, void, void, "/right[31:0]");
$add_traces(@nooverlay, void, void, "/leftAcc[31:0]");
$add_traces(@nooverlay, void, void, "/rightAcc[31:0]");
$add_traces(@nooverlay, void, void, "/delta[31:0]");
$add_traces(@nooverlay, void, void, "/refidx[9:0]");
$add_traces(@nooverlay, void, void, "/curidx[9:0]");
$add_traces(@nooverlay, void, void, "/matchidx[9:0]");
$add_traces(@nooverlay, void, void, "/limdata[31:0]");
$add_traces(@nooverlay, void, void, "/limaddr[3:0]");
// $add_traces(@nooverlay, void, void, "/wrLeft");
// $add_traces(@nooverlay, void, void, "/wrRight");
$add_traces(@nooverlay, void, void, "/assignTags");
$add_traces(@nooverlay, void, void, "/taggerInst/smstate");
$add_traces(@nooverlay, void, void, "/reload");
$add_traces(@nooverlay, void, void, "/addInterval");
$add_traces(@nooverlay, void, void, "/taggerInst/skipCnt");
$add_traces(@nooverlay, void, void, "/pulsestreamInst/outData1");
$add_traces(@nooverlay, void, void, "/pulsestreamInst/outData2");


$$setup_window_menu_item(20, 10, 10, 100, 400, 120, 40, "courier:12", @false, @decimal, @true, "Yellow",
void, "LightGray", "Black");
$$setup_environment_menu_item(80, 66, "", @charge, @decimal, @true, @true, @true, @true, void, void,
void, void, "time", 1e-09);
$run(200, @relative);
$force("numIntervals", "3", 0, void, void, @relative, @norepeat, void);
$force("MissingPulseThresh", "4", 0, void, void, @relative, @norepeat, void);
$force("maxT", "1182", 0, void, void, @relative, @norepeat, void);
$force("minMatchCnt", "1", 0, void, void, @relative, @norepeat, void);
$force("LimAddr", "0", 0, void, void, @relative, @norepeat, void);
$force("limData", "1174", 0, void, void, @relative, @norepeat, void);
$force("wrRight", "0", 0, void, void, @relative, @norepeat, void);
$force("wrLeft", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrLeft", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "0", 0, void, void, @relative, @norepeat, void);
```

P148505.PDF [Page: 104 of 108]


```
$run(100, @relative);
$select_point([-76, 279, "Trace"]);
$force("/LimData[31:0]", "1182", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limAddr", "1", 0, void, void, @relative, @norepeat, void);
$force("limdata", "873", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limdata", "881", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limdata", "958", 0, void, void, @relative, @norepeat, void);
$force("limaddr", "2", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limdata", "966", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limaddr", "3", 0, void, void, @relative, @norepeat, void);
$force("limdata", "827", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrleft", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limdata", "834", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "1", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("wrright", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("limdata", "100", 0, void, void, @relative, @norepeat, void);
$force("limdata", "100", 0, void, void, @relative, @norepeat, void);
```

P148505.PDF [Page: 105 of 108]


```
$$force_clock_menu_item("/clk", 100, 0, "1", 50, "0", void, @relative, @clear);
$run(100, @relative);
$force("reset", "0", 0, void, void, @relative, @norepeat, void);
$run(100, @relative);
$force("reset", "1", 0, void, void, @relative, @norepeat, void);
$run(2100, @relative);
```

UNCLASSIFI[redacted]

SECURITY CLASSIF[redacted] FORM
(highest classification of Ti[redacted], Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

LOCKHEED CANADA
1 IBER ROAD
STITTSVILLE, K2S 1E6 CANADA

2. SECURITY CLASSIFICATION (overall security classification of the document, including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

AN INVESTIGATION OF SPECIAL PURPOSE HARDWARE FOR FINE CHANNELIZATION IN THE AMEP PREPROCESSOR (U)

4. AUTHORS (Last name, first name, middle initial)

MICKEAL, J. AND MACKEAN, G.

5. DATE OF PUBLICATION (month and year of publication of document)

MARCH 1994

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

92

6b. NO. OF REFS (total cited in document)

3

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

CONTRACTOR'S FINAL REPORT

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
NATIONAL DEFENCE
SHIRLEYS BAY, OTTAWA, ONTARIO K1A 0K2 CANADA

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

011LB11

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written)

W7714-3-0119

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

1005058

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(XX Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

P148505.PDF [Page: 107 of 108]

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is h desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both offical languages unless the text is bilingual).

(U) The Advanced Modular ESM Processor (AMEP) uses a combination of special purpose hardware and a general purpose microprocessor to deinterleave Pulse Descript Word (PDW) data. This report investigates the possibility of increasing the proportion of processing performed in the special purpose hardware and thereby improving overall throughput.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selec from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible t select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

ELECTRONIC SUPPORT MEASURES (ESM)
FIELD PROGRAMMABLE GATE ARRAY
VHSIC HARDWARE DESCRIPTION LANGUAGE (VHDL)